ns-3 Testing and Validation

Release ns-3.10

ns-3 project

January 05, 2011

CONTENTS

This is the *ns-3 testing manual*. Primary documentation for the ns-3 project is available in four forms:

- ns-3 Doxygen: Documentation of the public APIs of the simulator
- Tutorial
- Reference Manual
- ns-3 wiki

This document is written in reStructuredText for Sphinx and is maintained in the `doc/testing` directory of ns-3's source code.

CHAPTER

ONE

OVERVIEW

This document is concerned with the testing and validation of *ns-3* software.

This document provides

- background about terminology and software testing (Chapter 2);
- a description of the ns-3 testing framework (Chapter 3);
- a guide to model developers or new model contributors for how to write tests (Chapter 4);

In brief, the first three chapters should be read by ns developers and contributors who need to understand how to contribute test code and validated programs, and the remainder of the document provides space for people to report on what aspects of selected models have been validated.

CHAPTER

TWO

BACKGROUND

This chapter may be skipped by readers familiar with the basics of software testing.

Writing defect-free software is a difficult proposition. There are many dimensions to the problem and there is much confusion regarding what is meant by different terms in different contexts. We have found it worthwhile to spend a little time reviewing the subject and defining some terms.

Software testing may be loosely defined as the process of executing a program with the intent of finding errors. When one enters a discussion regarding software testing, it quickly becomes apparent that there are many distinct mind-sets with which one can approach the subject.

For example, one could break the process into broad functional categories like ''correctness testing," ''performance testing," ''robustness testing" and ''security testing." Another way to look at the problem is by life-cycle: ''requirements testing," ''design testing," ''acceptance testing," and ''maintenance testing." Yet another view is by the scope of the tested system. In this case one may speak of ''unit testing," ''component testing," ''integration testing," and ''system testing." These terms are also not standardized in any way, and so ''maintenance testing" and ''regression testing" may be heard interchangeably. Additionally, these terms are often misused.

There are also a number of different philosophical approaches to software testing. For example, some organizations advocate writing test programs before actually implementing the desired software, yielding ''test-driven development." Some organizations advocate testing from a customer perspective as soon as possible, following a parallel with the agile development process: ''test early and test often." This is sometimes called ''agile testing." It seems that there is at least one approach to testing for every development methodology.

The *ns-3* project is not in the business of advocating for any one of these processes, but the project as a whole has requirements that help inform the test process.

Like all major software products, *ns-3* has a number of qualities that must be present for the product to succeed. From a testing perspective, some of these qualities that must be addressed are that *ns-3* must be ''correct," ''robust," ''performant" and ''maintainable." Ideally there should be metrics for each of these dimensions that are checked by the tests to identify when the product fails to meet its expectations / requirements.

2.1 Correctness

The essential purpose of testing is to determine that a piece of software behaves ''correctly." For *ns-3* this means that if we simulate something, the simulation should faithfully represent some physical entity or process to a specified accuracy and precision.

It turns out that there are two perspectives from which one can view correctness. Verifying that a particular model is implemented according to its specification is generically called *verification*. The process of deciding that the model is correct for its intended use is generically called *validation*.

2.2 Validation and Verification

A computer model is a mathematical or logical representation of something. It can represent a vehicle, an elephant (see David Harel's talk about modeling an elephant at SIMUTools 2009, or a networking card. Models can also represent processes such as global warming, freeway traffic flow or a specification of a networking protocol. Models can be completely faithful representations of a logical process specification, but they necessarily can never completely simulate a physical object or process. In most cases, a number of simplifications are made to the model to make simulation computationally tractable.

Every model has a *target system* that it is attempting to simulate. The first step in creating a simulation model is to identify this target system and the level of detail and accuracy that the simulation is desired to reproduce. In the case of a logical process, the target system may be identified as ''TCP as defined by RFC 793." In this case, it will probably be desirable to create a model that completely and faithfully reproduces RFC 793. In the case of a physical process this will not be possible. If, for example, you would like to simulate a wireless networking card, you may determine that you need, ''an accurate MAC-level implementation of the 802.11 specification and [...] a not-so-slow PHY-level model of the 802.11a specification.''

Once this is done, one can develop an abstract model of the target system. This is typically an exercise in managing the tradeoffs between complexity, resource requirements and accuracy. The process of developing an abstract model has been called *model qualification* in the literature. In the case of a TCP protocol, this process results in a design for a collection of objects, interactions and behaviors that will fully implement RFC 793 in *ns-3*. In the case of the wireless card, this process results in a number of tradeoffs to allow the physical layer to be simulated and the design of a network device and channel for ns-3, along with the desired objects, interactions and behaviors.

This abstract model is then developed into an *ns-3* model that implements the abstract model as a computer program. The process of getting the implementation to agree with the abstract model is called *model verification* in the literature.

The process so far is open loop. What remains is to make a determination that a given ns-3 model has some connection to some reality – that a model is an accurate representation of a real system, whether a logical process or a physical entity.

If one is going to use a simulation model to try and predict how some real system is going to behave, there must be some reason to believe your results – i.e., can one trust that an inference made from the model translates into a correct prediction for the real system. The process of getting the ns-3 model behavior to agree with the desired target system behavior as defined by the model qualification process is called *model validation* in the literature. In the case of a TCP implementation, you may want to compare the behavior of your ns-3 TCP model to some reference implementation in order to validate your model. In the case of a wireless physical layer simulation, you may want to compare the behavior of your model to that of real hardware in a controlled setting,

The *ns-3* testing environment provides tools to allow for both model validation and testing, and encourages the publication of validation results.

2.3 Robustness

Robustness is the quality of being able to withstand stresses, or changes in environments, inputs or calculations, etc. A system or design is ''robust" if it can deal with such changes with minimal loss of functionality.

This kind of testing is usually done with a particular focus. For example, the system as a whole can be run on many different system configurations to demonstrate that it can perform correctly in a large number of environments.

The system can be also be stressed by operating close to or beyond capacity by generating or simulating resource exhaustion of various kinds. This genre of testing is called ''stress testing.''

The system and its components may be exposed to so-called ''clean tests" that demonstrate a positive result – that is that the system operates correctly in response to a large variation of expected configurations.

The system and its components may also be exposed to ''dirty tests" which provide inputs outside the expected range. For example, if a module expects a zero-terminated string representation of an integer, a dirty test might provide an unterminated string of random characters to verify that the system does not crash as a result of this unexpected input. Unfortunately, detecting such ''dirty" input and taking preventive measures to ensure the system does not fail catastrophically can require a huge amount of development overhead. In order to reduce development time, a decision was taken early on in the project to minimize the amount of parameter validation and error handling in the *ns-3* codebase. For this reason, we do not spend much time on dirty testing – it would just uncover the results of the design decision we know we took.

We do want to demonstrate that *ns-3* software does work across some set of conditions. We borrow a couple of definitions to narrow this down a bit. The *domain of applicability* is a set of prescribed conditions for which the model has been tested, compared against reality to the extent possible, and judged suitable for use. The *range of accuracy* is an agreement between the computerized model and reality within a domain of applicability.

The *ns-3* testing environment provides tools to allow for setting up and running test environments over multiple systems (buildbot) and provides classes to encourage clean tests to verify the operation of the system over the expected ''domain of applicability" and ''range of accuracy.''

2.4 Performant

Okay, ''performant" isn't a real English word. It is, however, a very concise neologism that is quite often used to describe what we want *ns-3* to be: powerful and fast enough to get the job done.

This is really about the broad subject of software performance testing. One of the key things that is done is to compare two systems to find which performs better (cf benchmarks). This is used to demonstrate that, for example, *ns-3* can perform a basic kind of simulation at least as fast as a competing tool, or can be used to identify parts of the system that perform badly.

In the *ns-3* test framework, we provide support for timing various kinds of tests.

2.5 Maintainability

A software product must be maintainable. This is, again, a very broad statement, but a testing framework can help with the task. Once a model has been developed, validated and verified, we can repeatedly execute the suite of tests for the entire system to ensure that it remains valid and verified over its lifetime.

When a feature stops functioning as intended after some kind of change to the system is integrated, it is called generically a *regression*. Originally the term regression referred to a change that caused a previously fixed bug to reappear, but the term has evolved to describe any kind of change that breaks existing functionality. There are many kinds of regressions that may occur in practice.

A *local regression* is one in which a change affects the changed component directly. For example, if a component is modified to allocate and free memory but stale pointers are used, the component itself fails.

A *remote regression* is one in which a change to one component breaks functionality in another component. This reflects violation of an implied but possibly unrecognized contract between components.

An *unmasked regression* is one that creates a situation where a previously existing bug that had no affect is suddenly exposed in the system. This may be as simple as exercising a code path for the first time.

A *performance regression* is one that causes the performance requirements of the system to be violated. For example, doing some work in a low level function that may be repeated large numbers of times may suddenly render the system unusable from certain perspectives.

The *ns-3* testing framework provides tools for automating the process used to validate and verify the code in nightly test suites to help quickly identify possible regressions.

CHAPTER

THREE

TESTING FRAMEWORK

ns-3 consists of a simulation core engine, a set of models, example programs, and tests. Over time, new contributors contribute models, tests, and examples. A Python test program `test.py` serves as the test execution manager; `test.py` can run test code and examples to look for regressions, can output the results into a number of forms, and can manage code coverage analysis tools. On top of this, we layer *Buildbots* that are automated build robots that perform robustness testing by running the test framework on different systems and with different configuration options.

3.1 BuildBots

At the highest level of ns-3 testing are the buildbots (build robots). If you are unfamiliar with this system look at http://djmitche.github.com/buildbot/docs/0.7.11/. This is an open-source automated system that allows *ns-3* to be rebuilt and tested each time something has changed. By running the buildbots on a number of different systems we can ensure that *ns-3* builds and executes properly on all of its supported systems.

Users (and developers) typically will not interact with the buildbot system other than to read its messages regarding test results. If a failure is detected in one of the automated build and test jobs, the buildbot will send an email to the *ns-developers* mailing list. This email will look something like:

```
The Buildbot has detected a new failure of osx-ppc-g++-4.2 on NsNam.
Full details are available at:
 http://ns-regression.ee.washington.edu:8010/builders/osx-ppc-g%2B%2B-4.2/builds/0

Buildbot URL: http://ns-regression.ee.washington.edu:8010/

Buildslave for this Build: darwin-ppc

Build Reason: The web-page 'force build' button was pressed by 'ww': ww

Build Source Stamp: HEAD
Blamelist:

BUILD FAILED: failed shell_5 shell_6 shell_7 shell_8 shell_9 shell_10 shell_11 shell_12

sincerely,
-The Buildbot
```

In the full details URL shown in the email, one can search for the keyword `failed` and select the `stdio` link for the corresponding step to see the reason for the failure.

The buildbot will do its job quietly if there are no errors, and the system will undergo build and test cycles every day to verify that all is well.

3.2 Test.py

The buildbots use a Python program, `test.py`, that is responsible for running all of the tests and collecting the resulting reports into a human- readable form. This program is also available for use by users and developers as well.

`test.py` is very flexible in allowing the user to specify the number and kind of tests to run; and also the amount and kind of output to generate.

By default, `test.py` will run all available tests and report status back in a very concise form. Running the command

```
./test.py
```

will result in a number of `PASS`, `FAIL`, `CRASH` or `SKIP` indications followed by the kind of test that was run and its display name.

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
'build' finished successfully (0.939s)
FAIL: TestSuite ns3-wifi-propagation-loss-models
PASS: TestSuite object-name-service
PASS: TestSuite pcap-file-object
PASS: TestSuite ns3-tcp-cwnd
...
PASS: TestSuite ns3-tcp-interoperability
PASS: Example csma-broadcast
PASS: Example csma-multicast
```

This mode is indented to be used by users who are interested in determining if their distribution is working correctly, and by developers who are interested in determining if changes they have made have caused any regressions.

There are a number of options available to control the behavior of `test.py`. if you run `test.py --help` you should see a command summary like:

```
Usage: test.py [options]

Options:
  -h, --help            show this help message and exit
  -c KIND, --constrain=KIND
                        constrain the test-runner by kind of test
  -e EXAMPLE, --example=EXAMPLE
                        specify a single example to run
  -g, --grind           run the test suites and examples using valgrind
  -k, --kinds           print the kinds of tests available
  -l, --list            print the list of known tests
  -m, --multiple        report multiple failures from test suites and test
                        cases
  -n, --nowaf           do not run waf before starting testing
  -s TEST-SUITE, --suite=TEST-SUITE
                        specify a single test suite to run
  -v, --verbose         print progress and informational messages
  -w HTML-FILE, --web=HTML-FILE, --html=HTML-FILE
                        write detailed test results into HTML-FILE.html
  -r, --retain          retain all temporary files (which are normally
                        deleted)
  -t TEXT-FILE, --text=TEXT-FILE
                        write detailed test results into TEXT-FILE.txt
  -x XML-FILE, --xml=XML-FILE
                        write detailed test results into XML-FILE.xml
```

If one specifies an optional output style, one can generate detailed descriptions of the tests and status. Available styles are `text` and `HTML`. The buildbots will select the HTML option to generate HTML test reports for the nightly builds using

```
./test.py --html=nightly.html
```

In this case, an HTML file named ''nightly.html" would be created with a pretty summary of the testing done. A ''human readable" format is available for users interested in the details.

```
./test.py --text=results.txt
```

In the example above, the test suite checking the *ns-3* wireless device propagation loss models failed. By default no further information is provided.

To further explore the failure, `test.py` allows a single test suite to be specified. Running the command

```
./test.py --suite=ns3-wifi-propagation-loss-models
```

results in that single test suite being run.

```
FAIL: TestSuite ns3-wifi-propagation-loss-models
```

To find detailed information regarding the failure, one must specify the kind of output desired. For example, most people will probably be interested in a text file:

```
./test.py --suite=ns3-wifi-propagation-loss-models --text=results.txt
```

This will result in that single test suite being run with the test status written to the file ''results.txt''.

You should find something similar to the following in that file:

```
FAIL: Test Suite ''ns3-wifi-propagation-loss-models'' (real 0.02 user 0.01 system 0.00)
PASS: Test Case "Check ... Friis ... model ..." (real 0.01 user 0.00 system 0.00)
FAIL: Test Case "Check ... Log Distance ... model" (real 0.01 user 0.01 system 0.00)
  Details:
    Message:   Got unexpected SNR value
    Condition: [long description of what actually failed]
    Actual:    176.395
    Limit:     176.407 +- 0.0005
    File:      ../src/test/ns3wifi/propagation-loss-models-test-suite.cc
    Line:      360
```

Notice that the Test Suite is composed of two Test Cases. The first test case checked the Friis propagation loss model and passed. The second test case failed checking the Log Distance propagation model. In this case, an SNR of 176.395 was found, and the test expected a value of 176.407 correct to three decimal places. The file which implemented the failing test is listed as well as the line of code which triggered the failure.

If you desire, you could just as easily have written an HTML file using the `--html` option as described above.

Typically a user will run all tests at least once after downloading *ns-3* to ensure that his or her environment has been built correctly and is generating correct results according to the test suites. Developers will typically run the test suites before and after making a change to ensure that they have not introduced a regression with their changes. In this case, developers may not want to run all tests, but only a subset. For example, the developer might only want to run the unit tests periodically while making changes to a repository. In this case, `test.py` can be told to constrain the types of tests being run to a particular class of tests. The following command will result in only the unit tests being run:

```
./test.py --constrain=unit
```

Similarly, the following command will result in only the example smoke tests being run:

```
./test.py --constrain=unit
```

To see a quick list of the legal kinds of constraints, you can ask for them to be listed. The following command

```
./test.py --kinds
```

will result in the following list being displayed:

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
'build' finished successfully (0.939s)Waf: Entering directory `/home/craigdo/repos/ns-3-allinone-test
bvt:         Build Verification Tests (to see if build completed successfully)
core:        Run all TestSuite-based tests (exclude examples)
example:     Examples (to see if example programs run successfully)
performance: Performance Tests (check to see if the system is as fast as expected)
system:      System Tests (spans modules to check integration of modules)
unit:        Unit Tests (within modules to check basic functionality)
```

Any of these kinds of test can be provided as a constraint using the `--constraint` option.

To see a quick list of all of the test suites available, you can ask for them to be listed. The following command,

```
./test.py --list
```

will result in a list of the test suite being displayed, similar to:

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
'build' finished successfully (0.939s)
histogram
ns3-wifi-interference
ns3-tcp-cwnd
ns3-tcp-interoperability
sample
devices-mesh-flame
devices-mesh-dot11s
devices-mesh
...
object-name-service
callback
attributes
config
global-value
command-line
basic-random-number
object
```

Any of these listed suites can be selected to be run by itself using the `--suite` option as shown above.

Similarly to test suites, one can run a single example program using the `--example` option.

```
./test.py --example=udp-echo
```

results in that single example being run.

```
PASS: Example udp-echo
```

Normally when example programs are executed, they write a large amount of trace file data. This is normally saved to the base directory of the distribution (e.g., /home/user/ns-3-dev). When `test.py` runs an example, it really is completely unconcerned with the trace files. It just wants to to determine if the example can be built and run without

error. Since this is the case, the trace files are written into a `/tmp/unchecked-traces` directory. If you run the above example, you should be able to find the associated `udp-echo.tr` and `udp-echo-n-1.pcap` files there.

The list of available examples is defined by the contents of the ''examples" directory in the distribution. If you select an example for execution using the `--example` option, `test.py` will not make any attempt to decide if the example has been configured or not, it will just try to run it and report the result of the attempt.

When `test.py` runs, by default it will first ensure that the system has been completely built. This can be defeated by selecting the `--nowaf` option.

```
./test.py --list --nowaf
```

will result in a list of the currently built test suites being displayed, similar to:

```
ns3-wifi-propagation-loss-models
ns3-tcp-cwnd
ns3-tcp-interoperability
pcap-file-object
object-name-service
random-number-generators
```

Note the absence of the `Waf` build messages.

`test.py` also supports running the test suites and examples under valgrind. Valgrind is a flexible program for debugging and profiling Linux executables. By default, valgrind runs a tool called memcheck, which performs a range of memory- checking functions, including detecting accesses to uninitialised memory, misuse of allocated memory (double frees, access after free, etc.) and detecting memory leaks. This can be selected by using the `--grind` option.

```
./test.py --grind
```

As it runs, `test.py` and the programs that it runs indirectly, generate large numbers of temporary files. Usually, the content of these files is not interesting, however in some cases it can be useful (for debugging purposes) to view these files. `test.py` provides a `--retain` option which will cause these temporary files to be kept after the run is completed. The files are saved in a directory named `testpy-output` under a subdirectory named according to the current Coordinated Universal Time (also known as Greenwich Mean Time).

```
./test.py --retain
```

Finally, `test.py` provides a `--verbose` option which will print large amounts of information about its progress. It is not expected that this will be terribly useful unless there is an error. In this case, you can get access to the standard output and standard error reported by running test suites and examples. Select verbose in the following way:

```
./test.py --verbose
```

All of these options can be mixed and matched. For example, to run all of the ns-3 core test suites under valgrind, in verbose mode, while generating an HTML output file, one would do:

```
./test.py --verbose --grind --constrain=core --html=results.html
```

3.3 TestTaxonomy

As mentioned above, tests are grouped into a number of broadly defined classifications to allow users to selectively run tests to address the different kinds of testing that need to be done.

- Build Verification Tests
- Unit Tests
- System Tests

- Examples
- Performance Tests

3.3.1 BuildVerificationTests

These are relatively simple tests that are built along with the distribution and are used to make sure that the build is pretty much working. Our current unit tests live in the source files of the code they test and are built into the ns-3 modules; and so fit the description of BVTs. BVTs live in the same source code that is built into the ns-3 code. Our current tests are examples of this kind of test.

3.3.2 Unit Tests

Unit tests are more involved tests that go into detail to make sure that a piece of code works as advertised in isolation. There is really no reason for this kind of test to be built into an ns-3 module. It turns out, for example, that the unit tests for the object name service are about the same size as the object name service code itself. Unit tests are tests that check a single bit of functionality that are not built into the ns-3 code, but live in the same directory as the code it tests. It is possible that these tests check integration of multiple implementation files in a module as well. The file src/core/names-test-suite.cc is an example of this kind of test. The file src/common/pcap-file-test-suite.cc is another example that uses a known good pcap file as a test vector file. This file is stored locally in the src/common directory.

3.3.3 System Tests

System tests are those that involve more than one module in the system. We have lots of this kind of test running in our current regression framework, but they are typically overloaded examples. We provide a new place for this kind of test in the directory `src/test`. The file src/test/ns3tcp/ns3-interop-test-suite.cc is an example of this kind of test. It uses NSC TCP to test the ns-3 TCP implementation. Often there will be test vectors required for this kind of test, and they are stored in the directory where the test lives. For example, ns3tcp-interop-response-vectors.pcap is a file consisting of a number of TCP headers that are used as the expected responses of the ns-3 TCP under test to a stimulus generated by the NSC TCP which is used as a ‘’known good” implementation.

3.3.4 Examples

The examples are tested by the framework to make sure they built and will run. Nothing is checked, and currently the pcap files are just written off into /tmp to be discarded. If the examples run (don’t crash) they pass this smoke test.

3.3.5 Performance Tests

Performance tests are those which exercise a particular part of the system and determine if the tests have executed to completion in a reasonable time.

3.3.6 Running Tests

Tests are typically run using the high level `test.py` program. They can also be run manually using a low level test-runner executable directly from `waf`.

3.4 Running Tests Under the Test Runner Executable

The test-runner is the bridge from generic Python code to *ns-3* code. It is written in C++ and uses the automatic test discovery process in the *ns-3* code to find and allow execution of all of the various tests.

Although it may not be used directly very often, it is good to understand how `test.py` actually runs the various tests.

In order to execute the test-runner, you run it like any other ns-3 executable – using `waf`. To get a list of available options, you can type:

```
./waf --run "test-runner --help"
```

You should see something like the following:

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
'build' finished successfully (0.353s)
--assert:               Tell tests to segfault (like assert) if an error is detected
--basedir=dir:          Set the base directory (where to find src) to ''dir''
--tempdir=dir:          Set the temporary directory (where to find data files) to ''dir''
--constrain=test-type:  Constrain checks to test suites of type ''test-type''
--help:                 Print this message
--kinds:                List all of the available kinds of tests
--list:                 List all of the test suites (optionally constrained by test-type)
--out=file-name:        Set the test status output file to ''file-name''
--suite=suite-name:     Run the test suite named ''suite-name''
--verbose:              Turn on messages in the run test suites
```

There are a number of things available to you which will be familiar to you if you have looked at `test.py`. This should be expected since the test- runner is just an interface between `test.py` and *ns-3*. You may notice that example-related commands are missing here. That is because the examples are really not *ns-3* tests. `test.py` runs them as if they were to present a unified testing environment, but they are really completely different and not to be found here.

The first new option that appears here, but not in test.py is the `--assert` option. This option is useful when debugging a test case when running under a debugger like `gdb`. When selected, this option tells the underlying test case to cause a segmentation violation if an error is detected. This has the nice side-effect of causing program execution to stop (break into the debugger) when an error is detected. If you are using gdb, you could use this option something like,

```
./waf shell
cd build/debug/utils
gdb test-runner
run --suite=global-value --assert
```

If an error is then found in the global-value test suite, a segfault would be generated and the (source level) debugger would stop at the `NS_TEST_ASSERT_MSG` that detected the error.

Another new option that appears here is the `--basedir` option. It turns out that some tests may need to reference the source directory of the *ns-3* distribution to find local data, so a base directory is always required to run a test.

If you run a test from test.py, the Python program will provide the basedir option for you. To run one of the tests directly from the test-runner using `waf`, you will need to specify the test suite to run along with the base directory. So you could use the shell and do:

```
./waf --run "test-runner --basedir=`pwd` --suite=pcap-file-object"
```

Note the ''backward" quotation marks on the `pwd` command.

If you are running the test suite out of a debugger, it can be quite painful to remember and constantly type the absolute path of the distribution base directory. Because of this, if you omit the basedir, the test-runner will try to figure one out for you. It begins in the current working directory and walks up the directory tree looking for a directory file with files named `VERSION` and `LICENSE`. If it finds one, it assumes that must be the basedir and provides it for you.

Similarly, many test suites need to write temporary files (such as pcap files) in the process of running the tests. The tests then need a temporary directory to write to. The Python test utility (test.py) will provide a temporary file automatically, but if run stand-alone this temporary directory must be provided. Just as in the basedir case, it can be annoying to continually have to provide a `--tempdir`, so the test runner will figure one out for you if you don't provide one. It first looks for environment variables named `TMP` and `TEMP` and uses those. If neither `TMP` nor `TEMP` are defined it picks `/tmp`. The code then tacks on an identifier indicating what created the directory (ns-3) then the time (hh.mm.ss) followed by a large random number. The test runner creates a directory of that name to be used as the temporary directory. Temporary files then go into a directory that will be named something like

```
/tmp/ns-3.10.25.37.61537845
```

The time is provided as a hint so that you can relatively easily reconstruct what directory was used if you need to go back and look at the files that were placed in that directory.

When you run a test suite using the test-runner it will run the test quietly by default. The only indication that you will get that the test passed is the *absence* of a message from `waf` saying that the program returned something other than a zero exit code. To get some output from the test, you need to specify an output file to which the tests will write their XML status using the `--out` option. You need to be careful interpreting the results because the test suites will *append* results onto this file. Try,

```
./waf --run "test-runner --basedir=`pwd` --suite=pcap-file-object --out=myfile.xml"
```

If you look at the file `myfile.xml` you should see something like,

```
<TestSuite>
  <SuiteName>pcap-file-object</SuiteName>
  <TestCase>
    <CaseName>Check to see that PcapFile::Open with mode ''w'' works</CaseName>
    <CaseResult>PASS</CaseResult>
    <CaseTime>real 0.00 user 0.00 system 0.00</CaseTime>
  </TestCase>
  <TestCase>
    <CaseName>Check to see that PcapFile::Open with mode ''r'' works</CaseName>
    <CaseResult>PASS</CaseResult>
    <CaseTime>real 0.00 user 0.00 system 0.00</CaseTime>
  </TestCase>
  <TestCase>
    <CaseName>Check to see that PcapFile::Open with mode ''a'' works</CaseName>
    <CaseResult>PASS</CaseResult>
    <CaseTime>real 0.00 user 0.00 system 0.00</CaseTime>
  </TestCase>
  <TestCase>
    <CaseName>Check to see that PcapFileHeader is managed correctly</CaseName>
    <CaseResult>PASS</CaseResult>
    <CaseTime>real 0.00 user 0.00 system 0.00</CaseTime>
  </TestCase>
  <TestCase>
    <CaseName>Check to see that PcapRecordHeader is managed correctly</CaseName>
    <CaseResult>PASS</CaseResult>
    <CaseTime>real 0.00 user 0.00 system 0.00</CaseTime>
  </TestCase>
  <TestCase>
    <CaseName>Check to see that PcapFile can read out a known good pcap file</CaseName>
    <CaseResult>PASS</CaseResult>
```

```
    <CaseTime>real 0.00 user 0.00 system 0.00</CaseTime>
  </TestCase>
  <SuiteResult>PASS</SuiteResult>
  <SuiteTime>real 0.00 user 0.00 system 0.00</SuiteTime>
</TestSuite>
```

If you are familiar with XML this should be fairly self-explanatory. It is also not a complete XML file since test suites are designed to have their output appended to a master XML status file as described in the `test.py` section.

3.5 Class TestRunner

The executables that run dedicated test programs use a TestRunner class. This class provides for automatic test registration and listing, as well as a way to execute the individual tests. Individual test suites use C++ global constructors to add themselves to a collection of test suites managed by the test runner. The test runner is used to list all of the available tests and to select a test to be run. This is a quite simple class that provides three static methods to provide or Adding and Getting test suites to a collection of tests. See the doxygen for class `ns3::TestRunner` for details.

3.6 Test Suite

All *ns-3* tests are classified into Test Suites and Test Cases. A test suite is a collection of test cases that completely exercise a given kind of functionality. As described above, test suites can be classified as,

- Build Verification Tests
- Unit Tests
- System Tests
- Examples
- Performance Tests

This classification is exported from the TestSuite class. This class is quite simple, existing only as a place to export this type and to accumulate test cases. From a user perspective, in order to create a new TestSuite in the system one only has to define a new class that inherits from class `TestSuite` and perform these two duties.

The following code will define a new class that can be run by `test.py` as a ''unit" test with the display name, `my-test-suite-name`.

```
class MySuite : public TestSuite
{
public:
  MyTestSuite ();
};

MyTestSuite::MyTestSuite ()
  : TestSuite ("my-test-suite-name", UNIT)
{
  AddTestCase (new MyTestCase);
}

MyTestSuite myTestSuite;
```

The base class takes care of all of the registration and reporting required to be a good citizen in the test framework.

3.7 Test Case

Individual tests are created using a TestCase class. Common models for the use of a test case include "one test case per feature", and "one test case per method." Mixtures of these models may be used.

In order to create a new test case in the system, all one has to do is to inherit from the `TestCase` base class, override the constructor to give the test case a name and override the `DoRun` method to run the test.

```
class MyTestCase : public TestCase
{
  MyTestCase ();
  virtual bool DoRun (void);
};

MyTestCase::MyTestCase ()
  : TestCase ("Check some bit of functionality")
{
}

bool
MyTestCase::DoRun (void)
{
  NS_TEST_ASSERT_MSG_EQ (true, true, "Some failure message");
  return GetErrorStatus ();
}
```

3.7.1 Utilities

There are a number of utilities of various kinds that are also part of the testing framework. Examples include a generalized pcap file useful for storing test vectors; a generic container useful for transient storage of test vectors during test execution; and tools for generating presentations based on validation and verification testing results.

3.7.2 Debugging test suite failures

To debug test crashes, such as:

```
CRASH: TestSuite ns3-wifi-interference
```

You can access the underlying test-runner program via gdb as follows, and then pass the "–basedir=‘pwd‘" argument to run (you can also pass other arguments as needed, but basedir is the minimum needed):

```
./waf --command-template="gdb %s" --run "test-runner"
Waf: Entering directory ‘/home/tomh/hg/sep09/ns-3-allinone/ns-3-dev-678/build’
Waf: Leaving directory ‘/home/tomh/hg/sep09/ns-3-allinone/ns-3-dev-678/build’
’build’ finished successfully (0.380s)
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
L cense GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu"...
(gdb) r --basedir=‘pwd‘
Starting program: <..>/build/debug/utils/test-runner --basedir=‘pwd‘
[Thread debugging using libthread_db enabled]
```

```
assert failed. file=../src/core/type-id.cc, line=138, cond="uid <= m_information.size () && uid != 0"
...
```

Here is another example of how to use valgrind to debug a memory problem such as:

```
VALGR: TestSuite devices-mesh-dot11s-regression

./waf --command-template="valgrind %s --basedir=`pwd` --suite=devices-mesh-dot11s-regression" --run t
```

CHAPTER

FOUR

HOW TO WRITE TESTS

A primary goal of the ns-3 project is to help users to improve the validity and credibility of their results. There are many elements to obtaining valid models and simulations, and testing is a major component. If you contribute models or examples to ns-3, you may be asked to contribute test code. Models that you contribute will be used for many years by other people, who probably have no idea upon first glance whether the model is correct. The test code that you write for your model will help to avoid future regressions in the output and will aid future users in understanding the validity and bounds of applicability of your models.

There are many ways to test that a model is valid. In this chapter, we hope to cover some common cases that can be used as a guide to writing new tests.

4.1 Sample TestSuite skeleton

When starting from scratch (i.e. not adding a TestCase to an existing TestSuite), these things need to be decided up front:

- What the test suite will be called
- What type of test it will be (Build Verification Test, Unit Test, System Test, or Performance Test)
- Where the test code will live (either in an existing ns-3 module or separately in src/test/ directory). You will have to edit the wscript file in that directory to compile your new code, if it is a new file.

See the file `src/test/sample-test-suite.cc` and corresponding wscript file in that directory for a simple example, and see the directories under `src/test` for more complicated examples.

The rest of this chapter remains to be written

4.2 How to add an example program to the test suite

4.3 Testing for boolean outcomes

4.4 Testing outcomes when randomness is involved

4.5 Testing output data against a known distribution

4.6 Providing non-trivial input vectors of data

4.7 Storing and referencing non-trivial output data

4.8 Presenting your output test data