THE Automated Testing Framework

by Kristof Provost

There are several reasons to write tests, and if you only take away one idea from this article, it is that you should be writing tests because it's good for the soul. Okay, that might be a lie, but it's certainly good for the project.

Despite its importance, testing can be boring and repetitive work. Most of us are attracted to development work because we like building things more than we like breaking them. Fortunately, we have machines that are good at doing boring, repetitive work without ever asking for a raise or time off.

Automating tests has many advantages. It means repeatable tests. It means we know exactly what we are and are not testing, and when something goes wrong, we can usually easily reproduce the problem. Very few things make developers as happy as a bug report with a reproducible test case.

It also means it is easier to run the tests regularly, which means we more quickly discover things have broken. Users, it turns out, don't like it when their software suddenly explodes. Instead, they want it to be boring and predictable. Developers get to keep the exciting surprises to themselves.

How Do FreeBSD Tests Work?

Like all great artists, the FreeBSD developers stole/borrowed code from elsewhere. In this case, the source of inspiration was NetBSD. ATF, or the Automated Testing Framework, started out as a *Google Summer of Code* project. It was initially imported into NetBSD in 2007 and imported into FreeBSD in 2012.

ATF is the framework used to write tests. Another NetBSD project, Kyua, runs the tests and gathers up results. When running tests, you'll interact mostly with Kyua. When writing tests, you'll usually interact with ATF. It's possible to write tests without ATF and still have Kyua execute them, but we won't discuss that here.

Why Would I Want to Write a Test?

Good question, glad you asked!

There are several reasons to write tests, and if you only take one idea away from this article, it should be that you should be writing tests because it's good for the soul. Okay, that might be a lie, but it's certainly good for the project.

Tests ensure that functionality you rely on doesn't break. FreeBSD runs its test suite very frequently. The results of the test run for amd64 can be found on https://ci.freebsd.org/job/FreeBSD-head-amd64-test/.

Writing a test case is also a great way to help a developer fix your pet bug. A short test case that demonstrates the problem is often the key to getting bugs fixed. It not only conclusively demonstrates that there is a problem, but it also clearly explains how to reproduce it, which makes fixing it much easier.

When working on pf issues, I often find I spend much more time trying to understand the setup and trying to reproduce the issue than I do on the debugging and fixing. When bugs don't get fixed, it's often because they can't be reproduced.

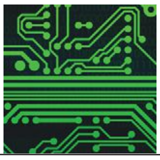
If you've got a pet patch you're trying to get committed, adding a test can also be helpful. It demonstrates the problem you're fixing or the new functionality you're adding. This serves both to convince people that this change is a good idea and as an aid in reviewing the patch. After all, a developer who commits a patch for you will have to fix any problems it might cause as well. Demonstrating that your code is tested will make it that much more likely that your patch will be included.

How to Use Tests

The tests, if installed, can be found in `/usr/tests` on your system. If you've also got a source tree, you can find the source code for the tests in `/usr/src/tests`, or in a subdirectory of the application, library or other they test. For example, tests for `/bin/sh` can be found in `/usr/src/bin/sh/tests`; `pfctl` tests can be found in `/usr/src/sbin/pfctl/tests`. Tests for pf itself can be found in `/usr/src/tests/sys/netpfil/pf`.

To run the tests, you'll want to install Kyua (`‘sudo pkg install kyua’`). You can then issue `‘kyua test‘` in `/usr/tests`. Many of the tests will want to be run as root:

```
/usr/tests % sudo kyua test
sbin/growfs/legacy_test:main  ->  passed  [4.710s]
sbin/devd/client_test:seqpacket  ->  passed  [0.015s]
sbin/devd/client_test:stream  ->  passed  [0.015s]
sbin/dhclient/option-domain-search_test:main  ->  passed  [0.005s]
sbin/pfctl/macro:space  ->  passed  [0.051s]
...
```



You can also run subsets of tests by running the command in the test subdirectory:

```
usr/tests/bin/cat % kyua test
cat_test:align  ->  passed  [0.023s]
cat_test:b_output  ->  passed  [0.025s]
cat_test:e_output  ->  passed  [0.023s]
cat_test:nonexistent  ->  passed  [0.022s]
cat_test:s_output  ->  passed  [0.026s]
cat_test:se_output  ->  passed  [0.023s]
cat_test:vt_output  ->  passed  [0.029s]
```

Alternatively, you can enumerate the list of tests using `'kyua list'` and pick individual tests to run:

```
/usr/tests % kyua test lib/csu/static/fini_test:dso_handle_test
lib/csu/static/fini_test:dso_handle_test -> passed [0.002s]
```

To get more information about tests, use `'kyua list -v'`. The extra information can include things like required programs, required user, or a description.

It goes without saying that this is something you'll only want to do on an otherwise unused system. By their very nature, tests risk exposing bugs, and some bugs are bad enough to bring the entire system down. Moreover, because the tests try to exercise all sorts of features of the system, it's quite possible they'll make configuration changes which can interfere with the intended use of the system.

Don't run them without permission from the admin of the system. If that happens to be you, the requirement to submit a written application in triplicate can probably be waived.

So, when would you run those tests? Usually during development, but they can also serve as smoke test for a newly installed system. Passing the tests does not guarantee that the hardware works perfectly, but it does increase confidence.

How to Write a Test

When it comes to writing your own tests, there are really two different topics to discuss: the mechanics of how to write the test and then the judgement of what to test, and how to test it.

We'll cover the mechanics first, briefly, as this is mostly documented in the `atf(7)` man page. Tests can be written as shell scripts or as C (or C++) code—which one is best depends on what is being tested. Usually the shell version (`'man 3 atf-sh'`) is best suited to test entire applications and the C/C++ version (`'man 3 atf-c`, `man 3 atf-c++'`) is best suited when testing libraries or (kernel) APIs.

atf-sh tests are executed by **atf-sh**, so they need a **#!/usr/bin/env atf-sh** first line. This is done by the installation code though, so we don't need to worry about it here. They always contain a **atf_init_test_cases** function. This lists the test cases. Each test case will always have a head and body and may also have a cleanup function:

```
atf_test_case tc1
tc1_head() {
    ... first test case's header ...
}
tc1_body() {
    ... first test case's body ...
}

atf_test_case tc2 cleanup
tc2_head() {
    ... second test case's header ...
}
tc2_body() {
    ... second test case's body ...
}
tc2_cleanup() {
    ... second test case's cleanup ...
}

... additional test cases ...

atf_init_test_cases() {
    atf_add_test_case tc1
    atf_add_test_case tc2
    ... add additional test cases ...
}
```

Note how all test case functions have the same prefix. This prefix is passed to the test system in the **atf_init_test_cases()** function, allowing the framework to find them. The **cleanup** function can have any name, but usually naming it **cleanup** is a good idea.

The head function is used to configure the test case. We can set the test description using '**atf_set "descr" "test description goes here"**'. If this test can only be run as a specific user (i.e., root) or on specific architectures, we'd add '**atf_set require.user root**' or '**atf_set require.arch amd64**'. A full list of options that can be set here can be found in '**man 4 atf-test-case**'.

If you need a more specific test, this can always be implemented

using ‘atf_skip‘. For example, tests that require the VIMAGE kernel option to be set can do this:

```
if [ "$(sysctl -i -n kern.features.vimage)" != 1 ]; then
        atf_skip "This test requires VIMAGE"
fi
```

One snag here is that constructs like that can't be used in the head. They'll need to go in the test body. Now that all of this setup and preparation is out of the way, we can finally look at the test body. This is where we run the application or code we want to put through its paces.

Most of the heavy lifting will be done by `atf-check(1)`. It executes a command and can help us by checking that the exit status, standard output, and/or standard error match our expectations. For example, we might want to test that the `false` command really returns the expected exit status (`-s exit:1`) and produces no output on either stdout (`-o empty`) or `stderr` (`-s empty`):

```
atf-check -s exit:1 -o empty -e empty /usr/bin/false
```

We could also expect a signal from the test program with `atf_expect_signal`, or expect a timeout with `atf_expect_timeout`.

Finally, if we've written a test for something we know to be broken, we can also add this information to the test. Using `atf_expect_fail` we can indicate that we expect this test to fail. Traditionally we'd also include the bug (PR) number in this message. The test will still be executed, but the failure won't be counted as a failed test.

C Tests

Tests in C (or C++) have a similar structure, albeit with somewhat different syntax.

Here's a short example:

```
#include <atf-c.h>

ATF_TC(tc1);
ATF_TC_HEAD(tc1, tc)
{
     atf_tc_set_md_var(tc, "require.user", "root");
}
ATF_TC_BODY(tc1, tc)
{
     ATF_CHECK_EQ(3, 2 + 1);
}
```

–continues next page–

–continued–

```
ATF_TP_ADD_TCS(tp)
{
     ATF_TP_ADD_TC(tcs, tc1);

     return atf_no_error();
}
```

The structure should be familiar after what we've covered already. We've got an entry point (**`ATF_TP_ADD_TCS()`**) that lists all of our test cases, and our test case has a head (**`ATF_TC_HEAD()`**) that tells the test framework we need to be root to run this test.

We may have lied about that last bit, because all our test body does is make sure that 2 + 1 equals 3. We'd generally expect that statement to be true, so this test should pass.

Hooking It All Up

Now that we know how to write tests, we still need to work out how we can actually run them. Fortunately, the makefiles do all of the hard work for us and we only have to write something like this:

```
# $FreeBSD$

ATF_TESTS_SH += sh_example
ATF_TESTS_C += c_example

.include <bsd.test.mk>
```

This would expect to find a 'sh_example.sh' file and 'c_example.c' file, build them, and install them as part of the 'buildworld' and 'install-world' make targets. The makefiles will also automatically create the required directory structure, and Kyuafile.

How to Debug Your Tests

A sad fact of life is that code is never perfect from the start (why else would we need tests?), and test code is no different. It's quite possible that your test won't do exactly what you expected it to do and the kyua output is typically not very helpful:

```
# kyua test example
example:example  ->  failed: atf-check failed; see the output
of the test for details  [0.017s]
```

Fortunately, kyua keeps track of a lot more than just that, and if we ask nicely, we can get more information about the failed test, its environment, and the output it generated during the tests:

```
# kyua report --verbose example:example
===> Execution context
Current directory: /usr/tests/sys/netpfil/pf
Environment variables:
     HOME=/root
     LANG=C
     LC_CTYPE=en_US.UTF-8
     LC_PAPER=en_GB.UTF-8
     LOGNAME=root
     MAIL=/var/mail/root
    PATH=/home/kp/bin:/usr/local/sbin:/sbin:/usr/local/bin:/
    bin:/usr/bin:/usr/sbin:/home/kp/bin:/bin:/sbin:/usr/bin:/
    usr/sbin:/usr/games:/usr/local/bin:/usr/local/sbin:/usr/
    pkg/bin:/usr/pkg/sbin
     SHELL=/bin/csh
     SUDO_COMMAND=/usr/local/bin/kyua test example
     SUDO_GID=1001
     SUDO_UID=1001
     SUDO_USER=kp
     TERM=xterm-256color
     USER=root
===> example:example
Result:      failed: atf-check failed;
see the output of the test for details
Start time: 2018-12-28T17:57:49.314738Z
End time:   2018-12-28T17:57:49.332141Z
Duration:   0.017s

Metadata:
     allowed_architectures is empty
     allowed_platforms is empty
     description is empty
     has_cleanup = false
     is_exclusive = false
     required_configs is empty
     required_disk_space = 0
     required_files is empty
     required_memory = 0
     required_programs is empty
     required_user is empty
     timeout = 300

Standard output:
Executing command [ false ]
```

–continues next page–

–continued–

```
Standard error:
Fail: incorrect exit status: 1, expected: 0
stdout:

stderr:

===> Failed tests
example:example  -> failed: atf-check failed; see the output of the test for details  [0.017s]
===> Summary
Results read from /root/.kyua/store/results.usr_tests_sys_netpfil_pf.20181228-175749-261396.db
Test cases: 1 total, 0 skipped, 0 expected failures, 0 broken, 1 failed
Start time: 2018-12-28T17:57:49.314738Z
End time:   2018-12-28T17:57:49.332141Z
Total time: 0.017s
```

The most interesting things here are the standard output and standard error logs. They tell us we were executing the command `'false'`, we expected error status `'0'`, but got `'1'` instead. In this case that's because our test is wrong. Naturally we'd expect `` `false` `` to exit with a status of 1 and not 0.

For more complex **`atf-sh`** tests, it can be quite useful to include `'set -x'` at the top of the test body. This will tell the shell to log everything it executes, giving a good overview of what commands were run and where the test stopped.

What Is a Good Test?

Knowing how to write the test is only half the battle. Knowing what to test is equally important. There are no hard rules about what makes for a good test, but there are a few tricks that can help to find good scenarios to test.

The easiest way is to start from a known bug. If you know something's broken, there's value in writing a test for it, if only to ensure that this particular bug will never happen again. Quite often, bugs will cluster around specific features or particular sections of code, so variations on that scenario may catch more bugs.

Another valuable test case is to check validation code. A lot of code checks user input to ensure that it falls within expected ranges. Pick a bunch of values and ensure that they're rejected or accepted as appropriate. Typically, very low or high values, as well as values just inside and just outside the valid range are good test points.

For example, say a given configuration knob is supposed to accept values between 0 and 10 inclusive. A good test would be to ensure that -1000, -1, 11, and 1000 are rejected and that 0, 5 and 10 are accepted.

When thinking about tests, one naturally tends to think about `'bad'` scenarios: bad input, bad configuration, bad everything. It's also useful to test a setup where everything works as expected.

Finally, it's important to remember that having some tests is vastly better than having none at all. Don't let uncertainty about what to test stop you. Write a test, any test, and you'll have contributed to the overall stability and quality of the system. Even a very basic test can reveal problems.

vnet

Finally, a few words about my favorite tests: network stack and firewall tests.

12.0 is the first release where vnet is enabled by default. I won't go into detail about vnet, that's a topic for a different article written by someone else.

vnet virtualizes the network stack. If that gobbledygook doesn't mean anything to you, try this: vnet gives each jail its own network stack. Network interfaces can belong exclusively to that jail. It can set its own IP addresses without help from the host. It's even possible to use ipfw or pf from inside the jail. A jail starts to look a lot like a virtual machine that way.

Long story short, it means that you can create network setups that'd usually require multiple machines all from the comfort of your laptop. In seconds. It means it's possible, and actually pretty easy, to write tests for the entire network stack. Speaking from my perfectly neutral and not-at-all-biased perspective as the pf maintainer, it means you should be writing tests for pf.

Find examples in `/usr/src/tests/sys/netpfil/pf`.

Other people have taken advantage of this example to write tests for IPSec, see `/usr/src/tests/sys/netipsec`. •

Kristof Provost is a freelance embedded software engineer specializing in network and video applications. He is a FreeBSD committer, maintainer of the pf firewall in FreeBSD, and a board member of the EuroBSDCon Foundation. Kristof has an unfortunate tendency to stumble into uClibc bugs and a burning hatred for FTP. Do not talk to him about IPv6 fragmentation.