Shaft DebConf9 Introduction

Adrián Pérez, Javier Muñoz <shaft@igalia.com>

In the present document, we introduce the reasoning for developing a high-level functional unit testing tool for shell code named Shaft (Shell Advanced Functional Tester) geared towards the GNU Bash shell. We also provide some key highlights of its advantages and features. The tool was developed at Igalia as part of the testing environment used in some of our projects.

Contents

1 Shell scripting in Debian

Every GNU/Linux system includes, as Debian does, utilities coded as shell scripts, due to its incomparable ability of blending existing Unix tools together. A great deal of system administration utilities including the boot system fall into this category.

Counting the lines in scripts run at system boot, which is a critical part of the system, there are more than 3.000 lines of code which are not being actively tested. Moreover, most packages do include small hook scripts which are run when they are installed and removed, summing up a total of 17K lines (10K if we do not take into account trivial scripts under 20 lines), and scanning the full file system goes over 85K lines of shell code[1].

Item	Lines
Initial RAM disk	1.200

[1] Tests done in a minimal Debian 5.0 "Lenny" fresh installation, line counts are physical (e.g. as done with `wc -l`).

Item	Lines
Init scripts	3.000
Package scripts (200 base packages)	17.300
Entire file system	88.000

There is an important amount of shell scripts which being verified using some kind of manual testing. Those scripts might be target of regressions so there would be a clear gain if the testing process is automated. As adding test suites for all of that code would be an extremely huge task, it would be nice to have have automated tests at least for the core system components.

2 Existing tools and techniques

2.1 Debugging

As far as we are concerned, there are two main techniques used for testing and debugging shell scripts, both requiring manual intervention.

The built-in command `set -x` makes the shell output expanded commands before they are executed, prepended by plus signs, one per each function nesting level. Although it is not a great deal of clarity, it serves its purpose when dealing with small amounts of code. Its main advantage is that it can be used with almost every Sh-compatible shell. There are two less-common switches which can serve as an extra aid: `set -v` will print each line read by the shell to standard error *before* executing it; and `set -u` will warn about usage of uninitialized variables. Unfortunately the last one introduces an annoying quirk: accessing an empty array will produce an error, which is definitely weird.

Of course, one can always do `echo`-based debugging, but there is not more we can do using using old-fashioned Sh-only features. Even the *Advanced Bash-Scripting Guide* advocates usage of `echo` statements although it briefly introduces the additional debugging features provided by Bash.

Bashdb is a nice debugger crafted by Rocky Bernstein, which does precisely use that extra features to implement a complete debugger for shell code. It can be used like any other debugger to control de execution of the code and inspecting the runtime environment. The same author is also responsible of incarnations of this piece of software for the ZSH and KSH shells.

2.2 Unit testing

Doing a quick search for `bash unit testing` renders a good amount of results, but most of the results deal with using shell script to test *other* kinds of software, not shell scripts *themselves*. There is, though, the one (and only) tool related to unit testing in this area: shUnit2, an unit testing framework inspired by the wave of the x-Unit family. Contrary to what Bashdb does, shUnit2 does not use any Bash-specific features, which is good so it can be used with hardly all the family of Sh-compatibles.

3 A better approach to testing

In our past experience, using just shUnit2 resulted in test suites which were filled up of lots of duplicated code. This poses several problems:

- Lengthy test suites become cumbersome and hard to understand.
- The amount of boilerplate code needed for each test case may be significantly more than the part used for testing.

[2] http://tldp.org/LDP/abs/html/debugging.html

- Creating new test cases may be slow because one has to think about all the logic driving the test, and not in just the test itself.

Taking those into account, and after some false starts trying to build upon shUnit2, we thought it would be interesting to develop a layer over the existing tools and simplify the work of those writing test suites. Our goals were clear:

- Each test case should be a separate text file (so ideally it would be self-contained).
- Test cases must be as declarative as possible (so they would be easy to write, read and understand).
- Provide a debugger-like approach to running shell code snippets in small steps, being able of inspecting the environment and making checks on-the-go. But without all the hassle of repeating boring tasks like setting breakpoints.
- Logging the output of the test cases as they are being run. This is useful e.g. for telling an user to run the test suite and then send a copy of the logs to the developer.

We added Bashdb in our toolchain to provide runtime introspection and execution control, but as it is more suitable for interactive use we had to add some glue: Shaft was born.

4 About the solution

4.1 The stack

The implementation of Shaft is based on the existing components:

- Bashdb is used to run the test cases in a controlled environment, and inspecting it as needed.
- shUnit2 already does a good job in counting test pass/fail rates, and provides some minimal support for initializing the environment and defining test suites.

Shaft is a layer above those components, which provides higher-level interface over them.

4.2 Defining tests the Shaft way

Test cases are written each in a separate text file which can include both shell functions and hooks for the system to use, and one-line commands. We call each test case in its separate file a *proof*.

The test suite is a small script which includes the main Shaft library, which and which must define some variables pointing to the script being tested and the directory containing the proof files. It will use that information to automagically build the test suite. Also, it will parse command line flags and enable logging if requested.

Shaft defines a superset of the Bashdb commands, so initially you may use any Bashdb command, as well the extended ones. An important set of extended commands are those which define checks.

Defining breakpoints in the code is automated by placing *break marks* in properly-formatted comments in the source code. Marks are textual tags with the same name as the file containing the proof, so a direct relation can be established between each test case and the piece of code being tested. Annotating the source code this way leaves room for improvement: as an example, a code coverage analysis tool could use them for determining which parts of the code are being actively tested by the existing proofs.

5 Final remarks

We believe that a tool like Shaft has some advantages over "traditional" testing methods, being the main one the ability to quickly add test cases. We feel like time should be spent doing interesting things, not repetitive tasks. Of course some degree of flexibility is lost when hiding the underlying tools, but it is a trade-off which worked well for us and that we think worths assuming in most cases.