Testing? What testing?

Peter Miller

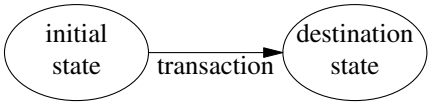
Platypus Technology

ABSTRACT

This paper presents a simplistic yet powerful model of what a test is. When you intend to test your software, you have to design your software to be test*able*. This paper will examine attributes of software implied by this model. Some examples of automated testing will be given.

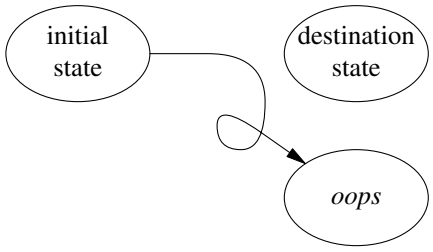
1. What is a test?

The core thesis of this paper is the idea[1] that a test consists of three things: a system in a defined state, a defined transaction, and a confirmation that the system arrives in a defined state.



This is an overly simplistic statement, but remains remarkable useful. The "system" under test could be a simple object, a collection of interrelated objects, a whole application, or a distributed multi-layer client-server system. Equally, the transaction could be a single byte of input, a single edge of a state transition diagram, or a series of transactions lumped together as a single event being considered.

Confirming that the system under test has arrived in a particular state can be done in may ways. Some states are clearly visable, sometimes they are available but not useful, and some internal states are not for user consumption and are much harder to access and therefore harder to confirm.



Please note that this is a *simplistic* definition of a test. It does not cover all forms of testing (such as tests of usability, maintainability, portability, robustness and so on which make up the other zillion software sub-characteristics listed in ISO 9126) and it is no substitute for a well thought out test plan. It does, however, provide some language for talking about functional testing.

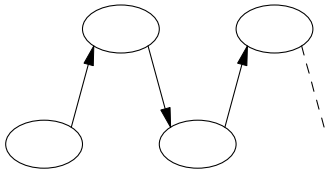
2. Manual testing is no testing

Humans are really bad at boring, repetitive tasks. If your test plan is based on the idea that your staff will faithfully execute a long list of printed instructions, at least once per release, then your testing is probably not effective.

For example, many manual test plans contain long sequences of things the operator is required to do, often with information on the screen to be confirmed as correct. This is all very well for successful tests, but what happens when one fails? Usually, these test scripts cover large numbers of behaviors. There is thus a motivation to complete the rest of the script, rather than stop, and have to do the start of the script again when the software has been fixed.

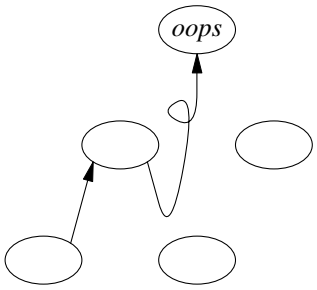
---

1. There is a growing body of knowledge called "Transaction Based testing" or sometimes "Transaction Based Verification".



There are two themes here: (a) testers have to look "productive" or they might not get paid, and (b) redoing the first bit again and again is boring.

Let's look at that definition again, rephrasing what our manual test scripts are doing. "Usually, these test scripts start from a defined state, and define a transaction and a confirmations of the destination state, then the next transaction and confirmation, *ad nauseum*." Now, what happens when one of those confirmations fails? Well, we know it's in *the wrong state*, so going on to execute the rest of the script, we are no longer fulfilling the initial portion of our three-part definition: we aren't in the defined state that the transaction is to be applied to. After the first failure, the rest of the results are *no information*.



For effective testing, then, you need something that is very good at accurately repeating the same script over and over again, and reporting very promptly when something goes wrong. Computers are very good at boring, repetitious tasks. They don't complain when you ask them to run the same stupid scripts tens or even thousands of times. And if the script breaks, they stop. For effective testing, then, you need automated testing. Let the humans *write* the tests, and let the computers *run* the tests.

3. Software Attributes

Automated testing requires the ability to automatically get the system under test into a defined state, the ability to automatically apply one or more transaction, and the ability to automatically confirm the current state (either read-and-compare, or write-and-diff, usually).

Some things are easy to test, e.g.

```
cat > test.in
cat > test.sed
cat > expected-output
sed-clone -f test.sed test.in \
    > test.out
diff expected-output test.out
```

But some things require some specific changes to get the three properties. *E.g.* a virtual machine simulator needs the ability to set registers and stack, *etc*, and later to dump them do they can be confirmed. This may be observable *e.g.* as some interesting opcodes only present in the simulator, and not the real machine, maybe to get the simulator to exit with a success/fail indicator.

3.1 Initial State

The system under test needs a way to be placed in a well defined initial state. This is something that most programs are reasonably good at. Word processors can load a file, image processing systems can load an image, databases can be created and populated with test sets, *etc*.

It was mentioned above that transactions can actually be a series of transactions. Sometimes, getting the system under test into a defined state requires starting from the default state and applying a series of known-to-work transactions. Provided that you can *get* the system under test into a defined state automatically, it can be tested automatically.

3.2 Transactions

Automating transactions can often be the hardest part of automated testing. Usually, this means automating the simulation of input. This could be user input, or a network connection, or a hardware simulation for an embedded application.

3.2.1 Command Line

The design of UNIX makes the testing of command line programs relatively simple, because you can redirect input from a file. This means that you don't actually need to change your software (or not much, anyway).

3.2.2 Full Screen

Full-screen programs are often similar, with input again directed from a file, although you may need to make it tolerant of non-tty input possibly under the control of a command line option. The trickier cases can be handled with *expect*.

3.2.3 GUI

On the other hand GUI interfaces are harder. There are some utilities, such as *TkReplay* which help. But they lead us to looking at the problem differently: where can we inject the input?
We can inject it into the X server (or have a fake X server which exists solely to provide test input).
We can proxy the X server, and inject the input via the proxy.
We can inject it into the event loop of our application. This, of course, requires changing the system under test.
We can have alternate input classes, a "real" one and an "automated" one. This, of course, means that the "real" input class doesn't get tested, but the rest of the system does, and that may be enough.

3.2.4 Client Server

Most of the techniques useful for X programs work for client server systems as well. Fake clients, fake servers, proxies, alternative input classes, *etc*.

3.2.5 Observation

In order to test the system, some aspect of it was changed. Auxiliary test support, more tolerant input, multiple input sources.

3.3 Verify State

Some programs, such as the *sed* example given above, are relatively easy to test. Many programs store a significant amount of state when you save to a file, and this may be compared with *diff*(1) or *cmp*(1). Other systems, however, are more challenging.

3.3.1 Full Screen

Many *curses*(3) programs need a special command to dump the screen into a text file for comparison using *diff*(1). It is also possible to use *expect* in many cases.

3.3.2 GUI

Many of the input solutions also work for output, but you will probably need special commands or options to get screen dumps at strategic moments, for comparison.

Wholesale capture and comparison of the output stream is problematic, usually because of gratuitous differences not relevant to the test.

3.3.3 Client Server

You can use bogus clients, bogus servers, or clever proxies.

3.3.4 Observation

In order to test the system, some aspect of it was changed. Auxiliary test support, captured output, multiple output destinations.

4. Discussion

There are some things which arise from consideration of these ideas.

4.1 No Result

In coming up with a testing regime, it is necessary to remember that tests do not simply *pass* or *fail*.

This is further complicated by the inverted sense of some tests. For example, your development process may require that a bug fix be accompanied by a test which *fails* on the unfixed system, and *passes* on the fixed system.

Consider the issues in achieving a necessary initial state by applying transactions to an initial state. What happens when one of these transactions, which are not the transaction under test, *fail*? In such a case it can't *fail*, because the bug fix case will give a false *positive*, but equally it can't succeed because this renders the test meaningless.

The solution is to have a third result, often called *no result*, which when negated still means *no result*.

Similar problems can occur with the transaction and verification stages of the test.

4.2 Negative Testing

Some other examples of negative testing will be given (i.e. *didn't* arrive in the right state, or invalid transactions resulting in an invalid state change).

4.3 Watch Me

A useful facility for creating tests is a "watch me" mode. This is a mode or tool or whatnot that allows the system to record inputs and output for replay and confirmation (respectively) at a later time. While this is *not* one of the necessary attributes, it is often a useful side effect.

4.4 Assert

This simple model of testing gives a different spin on the humble `assert` statement. The use of

`assert` can be thought of as verifying that the system is in a particular state, or that the transaction (input) is valid. This is not the kinf of artifact you *want* to see in production code; it is usually compiled out of production code.

4.5 Trace on Request

Another thing which is often compiled out of production code is a variety of tracing macros, which allow you to see the state of various portions of the system as they are executed. You sometimes see this in production systems, whene there is little performance impact; it is extremely useful feature for tech support, as well as testing.

5. Testing? What testing?

I once worked on an image processing system for which the company had partial source, and the inner workings where supplied as a library from the vendor. One of the transforms had some trouble, and I fixed it, but then I wondered how I should test it. How many of us can confirm visually that a 2D Walsh-Hadamard transform has worked correctly? While the destination state was visible on the screen, giving humans 2 side-by-side pictures (a "does it look like this" manual test) you will almost certainly get a false positive. *E.g.* those "find 10 differences" cartoon pictures on the funnies section of the newspaper. If humans are so bad at spotting *gross* differences, how can we expect them to find one pixel different in a million? So, I looked for the tool to compare two images and tell me how many pixels were different. *There wasn't one.* How did the vendor test their product?

If you have testability as a requirement of your software, you will write different software than if testability was not a requirement.

Do all the tools we use every day have these three properties: Can their initial state be loaded automatically? Can their transactions be applied automatically? Can their destination state be confirmed automatically? If any one of these is missing (but usually the last one), what gives us any confidence that they were tested at all?