Installed Tests Initiative

What is it?

Many tests run from “make check” are “black box” - they are designed to test the software’s external interfaces. There are numerous advantages to allowing them to be run separately from the build of the software.

How?

There are two major options. First, build the tests as part of the module’s normal build process, and install them to $prefix/tests/$PACKAGE/testname. Second, make the tests a separate ./configure;make;make install.

Why?

The current standard develop/test/deploy cycle that many “Linux distribution” type workflows encourage is this:

1) Developers edit source code in git repository
2) (optional) Subset of developers run “make check”
3) Developers reach stable point
4) One developer runs “make distcheck” on their laptop
5) Same developer uploads compressed tarball to FTP server
6) Packager inserts tarball into package
7) Package gets built in build server
8) (optional) Build server runs “make check”
9) Package is put on the Internet

The first problem with “distcheck” is that the developer’s laptop environment is not necessarily the same as the environment where it will actually be deployed. For example, an Ubuntu contributor working on pygobject may have a different version of Python from the person who is building it in OpenSUSE. Perhaps the biggest problem of this class is that developers will likely only build and run x86_64 binaries, but many distributions still ship 32 bit as well. It’s very easy for an uploaded source code tarball to be totally untested on 32 bit.

Then what’s the problem with running “make check” as part of builds in distribution build servers? While this is still significantly better than relying on ‘make distcheck’, the root problem still remains that the build environment is not necessarily the same as the target environment. Most distribution builders run in a “chroot” or equivalent, often on older (known stable) kernels. They may not have access to a system DBus instance, hardware GPUs, etc.

An even worse problem is if libfoo and bar (which depends on libfoo) are built at the same time. It's quite possible that bar is built (and tested) against an older libfoo which is in the build environment, then libfoo is built (and its tests pass), but it breaks bar. There is no way to run bar's tests without arbitrarily rebuilding bar (even though the source didn't change!), just to run tests.

The general principle here is that we want to test what's shipping to users, and that means the closer we are to testing **complete systems**, the better.

Installed tests are more powerful

The "distcheck" model constrains what tests can do, because the assumption is that they're running inside the same system that the developer's using. But with installed tests, we can do a lot more interesting things - particularly using virtualization (qemu), it's really easy to have destructive tests. We test things like deliberately injecting segfaults into gnome-shell, modify the system DBus configuration, adding/removing software, etc.

Further work

First, to really take advantage of this model, we'd need some standardized metadata for these installed tests. The first key bit of metadata is what kind of environment the test needs. For installed tests that originate as "distcheck" tests, we can assume they need a logged in desktop session as non root, and are non-destructive.

But if the metadata says a test needs to run as root and is destructive, we can simply snapshot+clone a VM, run the test, and shut down qemu afterwards with very little risk to the host system.

Possibly we could reuse .desktop files for this, or just a keyfile format.

Related art

http://anonscm.debian.org/gitweb/?p=autopkgtest/autopkgtest.git;a=blob_plain;f=doc/README.package-tests;hb=HEAD
http://autotest.github.com/
http://patches.openembedded.org/patch/32063/