Automating System Tests Using Declarative Virtual Machines

Sander van der Burg
Delft University of Technology
Delft, The Netherlands
s.vanderburg@tudelft.nl

Eelco Dolstra
Delft University of Technology
Delft, The Netherlands
e.dolstra@tudelft.nl

***Abstract*—Automated regression test suites are an essential software engineering practice: they provide developers with rapid feedback on the impact of changes to a system's source code. The inclusion of a test case in an automated test suite requires that the system's build process can automatically provide all the *environmental dependencies* of the test. These are external elements necessary for a test to succeed, such as shared libraries, running programs, and so on. For some tests (e.g., a compiler's), these requirements are simple to meet.**

However, many kinds of tests, especially at the integration or system level, have complex dependencies that are hard to provide automatically, such as running database servers, administrative privileges, services on external machines or specific network topologies. As such dependencies make tests difficult to script, they are often only performed manually, if at all. This particularly affects testing of distributed systems and system-level software.

This paper shows how we can automatically instantiate the complex environments necessary for tests by creating (networks of) virtual machines on the fly from declarative specifications. Building on NixOS, a Linux distribution with a declarative configuration model, these specifications concisely model the required environmental dependencies. We also describe techniques that allow efficient instantiation of VMs. As a result, complex system tests become as easy to specify and execute as unit tests. We evaluate our approach using a number of representative problems, including automated regression testing of a Linux distribution.

I. Introduction

Automated regression test suites are an essential software engineering practice, as they provide developers with rapid feedback on the impact of changes to a system's source code. By integrating such tests in the build process of a software project, developers can quickly determine whether a change breaks some functionality. These tests are easy to realise for certain kinds of software: for instance, a regression test for a compiler simply compiles a test case, runs it, and verifies its output; similarly, with some scaffolding, a unit test typically just calls some piece of code and checks its result.

However, other regression tests, particularly at the integration or system level, are significantly harder to automate because they have complex requirements on the environment in which the tests execute. For instance, they might require running database servers, administrative privileges, services on external machines or specific network topologies. This is especially a concern for distributed systems and system-level software. Consider for instance the following motivating examples used in this paper:

- *OpenSSH* is an implementation of the Secure Shell protocol that allows users to securely log in on remote systems. One component is the program `sshd`, the secure shell server daemon, which accepts connections from the SSH client program `ssh`, handles authentication, starts a shell under the requested user account, and so on. A test of the daemon must run with super-user privileges on Unix (i.e. as `root`) because the daemon must be able to change its user identity to that of the user logging in. It also requires the existence of several user accounts.
- *Quake 3 Arena* is a multiplayer first-person shooter video game. An automated regression test of the multiplayer functionality must start a server and a client and verify that the client can connect to the server successfully. Thus, such a test requires multiple machines. In addition, the clients (when running on Unix) require a running X11 server for their graphical user interfaces.
- *Transmission* is a Bittorrent client, managing downloads and uploads of files using the peer-to-peer Bittorrent protocol. Peer-to-peer operation requires that a running Bittorrent client is reachable by remote Bittorrent clients. This is not directly possible if a client resides behind a router that performs Network Address Translation (NAT) to map internal IP addresses to a single external IP address. Transmission clients automatically attempt to enable port forwarding in routers using the *UPnP-IGD* (Universal Plug and Play Internet Gateway Device) protocol. A regression test for this feature thus requires a network topology consisting of an IGD-enabled router, a client "behind" the router, and a client on the outside. The test succeeds if the second client can connect to the first through the router.

Thus, each of these tests requires special privileges, system services, external machines, or network topologies. This makes them difficult to include in an automated regression test suite: as these are typically started from (say) a Makefile on a developer's machine prior to checking in changes, or on a continuous build system, it is important that they are self-contained. That is, the test suite should set up the

complete environment that it requires. Without this property, test environments must be set up manually. For instance, we could manually set up a set of (virtual) machines to run the Transmission test suite, but this is labourious and inflexible. As a result, such tests tend to be done on an *ad hoc*, semi-manual basis. For example, many major system Unix packages (e.g. OpenSSH, the Linux kernel or the Apache web server) do not have automated test suites.

In this paper, we describe an approach to make such tests as easy to write and execute as conventional tests that do not have complex environmental dependencies. This opens a whole class of automated regression tests to developers. In this approach, a test *declaratively* specifies the environment necessary for the test, such as machines and network topologies, along with an imperative test script. From the specification of the environment, we can then automatically *build and instantiate virtual machines* (VMs) that implement the specification, and execute the test script in the VMs.

We achieve this goal by expanding on our previous work on *NixOS*, a Linux-based operating system distribution [1], which in turn builds on the purely functional package manager *Nix* [2]. In NixOS, the entire operating system – system packages such as the kernel, server packages such as Apache, end-user packages such as Firefox, configuration files in /etc and boot scripts, and so on – is built from source from a specification in what is in essence a purely functional "Makefile". (We give an overview of the relevant concepts in Nix and NixOS in Section II.) The fact that Nix builds from a purely functional specification means that configurations can easily be reproduced.

The latter aspect forms the basis for this paper. In a normal NixOS system, the configuration specification is used to build and activate a configuration on the local machine. In Section III, we show how these specifications can be used to produce the environment necessary for running a single-machine test. We also describe techniques that allow space and time-efficient instantiation of VMs. In Section IV, we extend this approach to networks of VMs. We discuss various aspects and applications of our work in Section V, including distributed code coverage analysis and continuous build systems. We have applied our approach to several real-world scenarios; we quantify this experience in Section VI.

II. Background: Nix and NixOS

In our approach, virtual machines are specified and built using the purely functional package manager *Nix*, and the VM instances that we build from the specifications are instances of NixOS, a Linux distribution based on Nix. In this section we give a brief overview of Nix and NixOS.

A. Nix

For the purposes of this paper, Nix [2] (http://nixos.org/) can be seen as a purely functional "Make". That is, like Make [3] and many other build tools, it performs build actions on the basis of a declarative specification of a graph of actions and their dependencies, but unlike Make, the specification is given in a lazy, purely functional language – the *Nix expression language*. This allows much more powerful abstractions to be expressed. Moreover, Nix stores the results of build actions such that they cannot interfere with each other, e.g. that the results of multiple invocations of a function do not overwrite each other. It stores the output of a build step, or *derivation*, under a unique path such as

/nix/store/q325djkc1ivlfyzan22197dc62gbq04z-firefox-3.5

where q325djkc1ivl... is a cryptographic hash of the inputs of the derivation, such as sources, compilers, libraries and build scripts.

The fundamental operation in the Nix expression language is the built-in function derivation, which takes as argument a set of name/value pairs, or *attributes*:

```
derivation {
  name = "foo";
  builder = "${bash}/bin/sh";
  args = [ "-c" "echo Hello $who > $out" ];
  who = "world";
}
```

A derivation describes the invocation of a command (usually a shell script) that must produce output under a path in the Nix store. The derivation is built by executing a program, whose path and command-line arguments are specified in the attributes builder and args, respectively. The other attributes are passed to the builder as environment variables. Attribute values can be (lists of) strings or other derivations. The latter denote the dependencies of the current derivation. When building a derivation, its dependencies are built first. The path of each dependency's output in the Nix store is placed in the corresponding environment variable. Strings can also contain references to other derivations, enclosed in ${...}. These are replaced by the derivation's output path in the Nix store.

For instance, if we evaluate the foo derivation above, first the derivation denoted by the variable bash (not shown here) is built, resulting in a store path like /nix/store/49ndfiqrlc9b...-bash-4.0-p17. Then foo is built, with the environment variable who set to world. Nix passes the intended location of the output in the Nix store, computed by hashing the input attributes, through the environment variable out. Thus, the derivation above will write the string Hello world to a path such as /nix/store/6dsdb0j20n3b...-foo.

A derivation can build anything, as long as it is pure, i.e. depends only on its explicitly defined inputs, and produces output under the path denoted by the environment variable out. Nix is primarily intended as a deployment tool – a *package manager*. Thus derivations are typically large steps that build entire packages. Figure 1 shows an example of a Nix expression to build the Apache web server. The language construct rec { ... } defines a set of variable

```
rec {
  httpd = stdenv.mkDerivation {
    name = "apache-httpd-2.2.13";
    src = fetchurl {
      url = http://.../httpd-2.2.13.tar.bz2;
      md5 = "8d8d904e7342125825ec70f03c5745ef";
    };
    buildInputs =
      [ perl apr aprutil pcre openssl ];
    configureFlags =
      "--enable-mods-shared=all ...";
  };

  apr = stdenv.mkDerivation {
    name = "apr-1.3.8"; ...
  };

  stdenv.mkDerivation = args: derivation {
    builder = ...
      ''
        PATH=${gcc}/bin:${coreutils}/bin:...
        tar xf ${args.src}
        ./configure --prefix=$out \
          ${args.configureFlags}
        make
        make install
      ''; ...
  };
  ...
}
```

Figure 1. pkgs.nix: Nix expression to build Apache

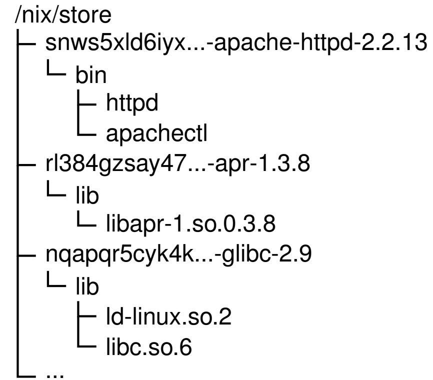

Figure 2. Result of building Apache in the Nix store

```
{ config, pkgs, ... }:
{
  services.httpd.enable = true;
  services.httpd.documentRoot = "/www-root";
  services.xserver.enable = true;
  services.desktopManager.kde4.enable = true;
  environment.systemPackages = [ pkgs.firefox ];
}
```

Figure 3. NixOS configuration module

bindings that can refer to each other, e.g. httpd refers to apr (the derivation that builds the Apache runtime package).

The derivation httpd shows the use of function abstractions to capture common build patterns: it calls the function stdenv.mkDerivation, which performs a build of a standard Unix-style package (namely, unpack the source, run an Autoconf configure script, run make to build, and finally make install to install the package under $out). Functions are defined using the syntax arg: body. Functions can also pattern-match on attribute sets: a function $\{\text{arg}_1, ..., \text{arg}_n\}$: body must be called with an attribute set containing the named attributes. Ellipses can be used in the argument list to denote that additional attributes are to be ignored.

We can build Apache from the command line as follows:

```
$ nix-build pkgs.nix -A httpd
```

This builds the attribute httpd from the file pkgs.nix in Figure 1, after building its dependencies such as perl, apr and gcc. The result of building Apache on the Nix store is seen in Figure 2.

There is a large distribution of Nix expressions, the *Nix Packages collection*, that contains almost 2500 packages, and supports a variety of operating systems.

B. NixOS

Nix has been used to build a Linux distribution, NixOS [1]. NixOS uses Nix to build the entire system from a specification in the Nix expression language – not just software packages. *All* static parts of the system – packages, the kernel, boot scripts, scripts to manage system services, configuration data, and so on – are built by Nix derivations. In fact, there is a single top-level derivation that, when built, causes all static parts of the system to be built as dependencies. The practical advantages of a purely functional approach to system configuration management are that upgrading the system is safe (since the old configuration in the Nix store is not overwritten) and reliable (since due to purity it does not rely on the previous state of the system), we can always roll back to previous configurations, and we can deterministically rebuild a configuration.

NixOS has a declarative configuration model. The Nix expressions that constitute NixOS are organised into *modules* that together build the system. NixOS currently consists of around 125 modules, each implementing some part of the system (e.g. building the boot scripts, the Apache configuration, or the X11 GUI environment). In addition, the end-user configuration of a NixOS machine is also specified as a module. Figure 3 shows an example of a NixOS module specifying the high-level configuration of a system. It states that the system should run Apache to serve files in the directory /www-root, have a graphical user interface running the KDE desktop environment, and provide the Firefox web browser to users. Other modules compute values that depend on this configuration. For instance, the values of the attributes services.httpd.enable and services.httpd.documentRoot are used by another module – the Apache web server module – to determine whether to

generate a script to start and manage Apache, as well as the contents of its configuration file httpd.conf.

The basic structure of a NixOS module is:

```
{ config, pkgs, ... }:
{ ... configuration values ... }
```

That is, a module is a function that accepts at least two arguments: config, which contains the full system configuration, and pkgs, which contains the Nix Packages collection for convenience. For instance, the value pkgs.httpd is the derivation that builds the Apache web server. The system configuration config is computed by calling every NixOS module and merging the attribute sets of configuration values returned by each. The result of the merge is passed back as the config function argument to each module. (This is possible because the Nix expression language is lazy.)

Thus each module contributes values to the set config and can use values defined by other modules. Most configuration values are system options relevant to end users, but others are "computed" values that are derived from other configuration values. For instance, the value of the attribute build.system.kernel is a derivation that builds the Linux kernel. The entire system is built by the attribute build.system.toplevel, whose value is a derivation that has all other parts of the system as dependencies. Thus, the following command builds the entire operating system, including all packages, scripts, configuration files and system services:

```
$ nix-build /etc/nixos/nixos \
    -A config.build.system.toplevel
```

The important property here is that NixOS provides us with a way to deterministically and automatically build a complete operating system environment, with all its dependencies, from a declarative specification. As we shall see in the next section, we can define other "top-level" derivations that instantiate virtual machines from a configuration, and extend the single-machine specifications such as the one in Figure 3 to networks of machines.

III. Single-machine tests

NixOS system configurations allow developers to concisely specify the environment necessary for a integration or system test and instantiate virtual machines from these, even if such tests require special privileges or running system services. After all, inside a virtual machine, we can do any actions that would be dangerous or not permitted on the host machine. We first address single-machine tests; in the next section, we extend this to networks of machines.

A. Specifying and running tests

Figure 4 shows an implementation of the OpenSSH regression test described in Section I. It consists of two parts: a declarative specification of the machine in which the test is to be performed (the attribute machine), and an imperative

```
let openssh = stdenv.mkDerivation { ... }; in
makeTest {
  machine =
    { config, pkgs, ... }:
    { users.extraUsers =
        [ { name = "sshd"; home = "/var/empty"; }
          { name = "bob";  home = "/home/bob";  }
        ];
    };

  testScript = ''
    $machine→succeed(
      "${openssh}/bin/ssh-keygen " .
         "-f /etc/ssh/ssh_host_dsa_key",
      "${openssh}/sbin/sshd -f /dev/null",
      "mkdir -m 700 /root/.ssh /home/bob/.ssh",
      "${openssh}/bin/ssh-keygen " .
        "-f /root/.ssh/id_dsa",
      "cp /root/.ssh/id_dsa.pub " .
        "/home/bob/.ssh/authorized_keys");
    $machine→waitForOpenPort(22);
    $machine→succeed("${openssh}/bin/ssh " .
      "bob\@localhost 'echo \$USER'")
      eq "bob\n" or die;
  '';
}
```

Figure 4. openssh.nix: Specification of an OpenSSH regression test

test script (testScript). The machine specification is very simple: all that we need for the test beyond a basic NixOS machine is the existence of two user accounts (sshd for the SSH daemon's *privilege separation* feature, and bob as a test account for logging in).

The test script is a Perl script running on the host that performs operations in the virtual machine (the *guest*) using a number of primitives. For instance, succeed executes shell commands in the guest and aborts the test if they fail, while waitForOpenPort waits until the guest is listening on the specified TCP port. The OpenSSH test script creates an SSH *host key* (required by the daemon to allow clients to verify that they are connecting to the right machine), starts the daemon, creates a public/private key pair, add the public key to Bob's list of allowed keys, waits until the daemon is ready to accept connections, and logs in as Bob using the private key. Finally, it verifies that the SSH session did indeed log in as Bob and signals failure otherwise.

The function makeTest applied to machine and testScript evaluates to two attributes: vm, a derivation that builds a script to starts a NixOS VM matching the specification in machine; and test, a derivation that depends on vm, runs its script to start the VM, and then executes testScript. Thus, the following command performs the OpenSSH test:

```
$ nix-build openssh.nix -A test
```

That is, it builds the OpenSSH package as well as a complete NixOS instance with its hundreds of dependencies. For interactive testing, a developer can also do:

```
$ nix-build openssh.nix -A vm
```

```
$ ./result/bin/run-vm
```

(The call to nix-build leaves a symbolic link result to the output of the vm derivation in the Nix store.) This starts the virtual machine on the user's desktop, booting a NixOS instance with the specified functionality.

B. Implementation

Two important practical advantages of our approach are that the implementation of the VM requires no root privileges and is self-contained (openssh.nix and the expressions that build NixOS completely describe how to build a VM automatically). These properties allow such tests to be included in an automated regression test suite.

Virtual machines are built by the NixOS module qemu-vm.nix that defines a configuration value system.build.vm, which is a derivation to build a shell script that starts the NixOS system built by system.build.toplevel in a virtual machine. We use QEMU/KVM (http://www.linux-kvm.org/), a modified version of the open source QEMU processor emulator that uses the hardware virtualisation features of modern CPUs to run VMs at near-native speed. An important feature of QEMU over most other VM implementations is that it allows VM instances to be easily started and controlled from the command line. This includes the fully automated starting, running and stopping of a VM in a derivation. Furthermore, QEMU provides special support for booting Linux-based operating systems: it can directly boot from a kernel and initial ramdisk image on the host filesystem, rather than requiring a full hard disk image with that kernel installed. (The initial ramdisk in Linux is a small filesystem image responsible for mounting the real root filesystem.) For instance, the system.build.vm derivation generates essentially this script:

```
${pkgs.qemu_kvm}/bin/qemu-system-x86_64 -smb /
 -kernel ${config.boot.kernelPackages.kernel}
 -initrd ${config.system.build.initialRamdisk}
 -append "init=${config.system.build.bootStage2}
  systemConfig=${config.system.build.toplevel}"
```

The system.build.vm derivation does not build a virtual hard disk image for the VM, as is usual. Rather, the initial ramdisk of the VM mounts the Nix store of the host through the network filesystem *CIFS* (the -smb / option above); QEMU automatically starts a CIFS server on the host to service requests from the guest. This is a crucial feature: the set of dependencies of a system is hundreds of megabytes in size at the least, so to build such an image every time we reconfigure the VMs would be very wasteful in time and space. Thus, rebuilding a VM after a configuration change usually takes only a few seconds.

The VM start script does create an empty ext3 root filesystem for the guest at startup, to hold mutable state such as the contents of /var or the system account file /etc/passwd. Thanks to sparse allocation of blocks in the virtual disk image, image creation takes a fraction of a second. NixOS's boot process is self-initialising: it initialises all state needed to run the system. For interactive use, the filesystem is preserved across restarts of the VM, saved in the image file ./*hostname*.qcow2.

QEMU provides virtualised network connectivity to VMs. The VM has a network interface, eth0, that allows it to talk to the host. The guest has IP address 10.0.2.15, QEMU's virtual gateway to the host is 10.0.2.2, and the CIFS server is 10.0.2.4. This feature is implemented entirely in user space: it requires no root privileges.

The test script executes commands on the VM using a root shell running on the VM that receives commands from the host through a TCP connection between the VM and the host. (The remotely accessible root shell is provided by a NixOS module added to the machine configuration by makeTest and does not exist in normal use.) QEMU allows TCP ports in the guest network to be forwarded to Unix domain sockets [4] on the host. This is important for security: we do not want anybody other than the test script connecting to the port. Also, in continuous build environments any number of builds may execute concurrently; the use of fixed TCP ports on the host would preclude this.

IV. Distributed tests

Many typical system tests are distributed: they require multiple machines to execute. Therefore a natural extension of the declarative machine specifications in the previous section is to specify entire *networks* of machines, including their topologies.

A. Specifying networks

Figure 5 shows a small automated test for Quake 3 Arena. As described in Section I, it verifies that clients can successfully join a game running on a non-graphical server. Here makeTest is called with a nodes argument instead of a single machine. This is a set specifying all the machines in the network; each attribute is a NixOS configuration module. In this case, it specifies a network of two machines: server, which automatically starts a Quake server daemon, and client, which runs an X11 graphical user interface and has the Quake client installed, but otherwise does nothing. The attribute test returned by makeTest evaluates to a derivation that executes the VMs in a virtual network and runs the given test suite. The test script uses additional test primitives, such as waitForJob (which waits until the given system service has started successfully) and screenshot (which takes a screenshot of the virtual display of the VM).

The test script in the example first starts all machines and waits until they are ready. This speeds up the test as it boots the machines in parallel; otherwise they are only booted on demand, i.e., when the test script performs an action on the machine. It then executes a command on the client to start a graphical Quake client and connect to the server. After a while, we verify on the server that the client did indeed

```
makeTest {
  nodes =
    { server =
        { config, pkgs, ... }:
        { jobs.quake3Server =
            { startOn = "startup";
              exec =
                "${pkgs.quake3demo}/bin/quake3"
                + " +set dedicated 1"
                + " +set g_gametype 0"
                + " +map q3dm7 +addbot grunt"
                + " 2> /tmp/log";
            };
        };
      client =
        { config, pkgs, ... }:
        { services.xserver.enable = true;
          environment.systemPackages =
            [ pkgs.quake3demo ];
        };
    };

  testScript = ''
    startAll;
    $server→waitForJob("quake3-server");
    $client→waitForX;
    $client→succeed(
      "quake3 +set name Foo +connect server &");
    sleep 40;
    $server→succeed(
      "grep 'Foo.*entered the game' /tmp/log");
    $client→screenshot("screen.png");
  '';
}
```

Figure 5. Specification of a Quake client/server regression test

connect. The derivation will fail to build if this is not the case. Finally, we make a screenshot of the client to allow visual inspection of the end state, if desired.

GUI testing is a notoriously difficult subject [5]. The point here is not to make a contribution to GUI testing techniques per se, but to show that we can easily set up the infrastructure needed for such tests. In the test script, we can run any desired automated GUI testing tool.

The virtual machines can talk to each other because they are connected together into a virtual network. Each VM has a network interface eth1 with an IP address in the private range 192.168.1.n assigned in sequential order by makeTest. (Recall that each VM also has a network interface eth0 to communicate with the host.) QEMU propagates any packet sent on this interface to all other VMs in the same virtual network. The machines are assigned hostnames equal to the corresponding attribute name in the model, so the machine built from the server attribute has hostname server.

B. Complex topologies

The Quake test has a trivial network topology: all machines are on the same virtual network. The test of the port forwarding feature in the Transmission Bittorrent client (described in Section I) requires a more complicated topology:

```
nodes = {
  tracker =
    { config, pkgs, ... }:
    { environment.systemPackages =
        [ pkgs.transmission pkgs.bittorrent ];
      services.httpd.enable = true;
      services.httpd.documentRoot = "/tmp";
    };
  router =
    { config, pkgs, ... }:
    { environment.systemPackages =
        [ iptables miniupnpd ];
      virtualisation.vlans = [ 1 2 ];
    };
  client1 =
    { config, pkgs, nodes, ... }:
    { environment.systemPackages = [transmission];
      virtualisation.vlans = [ 2 ];
      networking.defaultGateway = nodes.router
        .config.networking.ifaces.eth2.ipAddress;
    };
  client2 =
    { config, pkgs, ... }:
    { environment.systemPackages = [transmission];
    };
};
```

Figure 6. Network specification for the Transmission regression test

an "inside" network, representing a typical home network behind a router, and an "outside" network, representing the Internet. The router should be connected to both networks and provide Network Address Translation (NAT) from the inside network to the outside. Machines on the inside should not be directly reachable from the outside. Thus, we cannot do this with a single virtual network. To support such scenarios, makeTest can create an arbitrary number of virtual networks, and allows each machine specification to declare in the option virtualisation.vlans to what networks they should be connected.

Figure 6 shows the specification of the machines for the Transmission test. It has two virtual networks, identified as 1 (the "outside" network) and 2 (the "inside"). There are four machines: router is connected to both, client1 is connected to 2, while tracker and client2 are connected to 1. (If virtualisation.vlans is omitted, it defaults to 1.) The tracker runs the Apache web server to make torrent files available to the clients. The configuration further specifies what packages should be installed on what machines, e.g., the router needs the iptables and miniupnpd packages for its NAT and UPnP-IGD functionality.

The test, shown in Figure 7, proceeds as follows. We first initialise NAT on the router. We then create a torrent file on the tracker and start the tracker program, a central Bittorrent component that keeps track of the clients that are sharing a given file, on port 6969. Also on the tracker we start the *initial seeder*, a client that provides the initial copy of the file so that other clients can obtain it. We then start a download on the client behind the router and wait until it finishes.

```
testScript = ''
  # Enable NAT on the router and start miniupnpd.
  $router→succeed(
    "iptables -t nat -F", ...
    "miniupnpd -f ${miniupnpdConf}");

  # Create the torrent and start the tracker.
  $tracker→succeed(
    "cp ${file} /tmp/test",
    "transmissioncli -n /tmp/test /tmp/test.torrent",
    "bittorrent-tracker --port 6969 &");
  $tracker→waitForOpenPort(6969);

  # Start the initial seeder.
  my $pid = $tracker→background(
    "transmissioncli /tmp/test.torrent -w /tmp");

  # Download from the first (NATted) client.
  $client1→succeed("transmissioncli " .
    "http://tracker/test.torrent -w /tmp &");
  $client1→waitForFile("/tmp/test");

  # Bring down the initial seeder.
  $tracker→succeed("kill -9 $pid");

  # Now download from the second client.
  $client2→succeed("transmissioncli " .
    "http://tracker/test.torrent -w /tmp &");
  $client2→waitForFile("/tmp/test");
'';
```

Figure 7. Test script for the Transmission regression test

If Transmission and miniupnpd work correctly in concert, the router should now have opened a port forwarding that allows the second client to connect to the first client. To verify that this is the case, we shut down the initial seeder and start a download on the second client. This download can only succeed if the first client is reachable through the NAT router.

Each virtual network is implemented as a separate QEMU network; thus a VM cannot send packets to a network to which it is not connected. Machines are assigned IP addresses 192.168.$n.m$, where n is the number of the network and m is the number of the machine, and have Ethernet interfaces connected to the requested networks. For example, the router will have interfaces eth1 with IP address 192.168.1.3 and eth2 with address 192.168.2.3, while the first client will only have an interface eth1 with IP address 192.168.2.1. The test infrastructure provides operations to simulate events such as network outages or machine crashes.

V. Discussion

Declarative model: To what extent do we need the properties of Nix and NixOS, in particular the fact that an entire operating system environment is built from source from a specification in a single formalism, and the purely functional nature of the Nix store? There are many tools to automate deployment of machines. For instance, Red Hat's Kickstart tool installs RPM-based Linux systems from a textual specification and can be used to create virtual machines automatically, with a single command-line invocation [6].

However, there are many limitations to such tools:

- Having a single formalism that describes the construction of an entire network from source makes hard things easy, such as building part of the system with coverage analysis. In a tool such as Kickstart, the binary software packages are a given; we cannot easily modify the build processes of those packages.
- For testing in virtual machines, it is important that VMs can be built efficiently. With Nix, this is the case because the VM can use the host's Nix store. With other package managers, that is not an option because the host filesystem may not contain the (versions of) packages that a VM needs. One would also need to be root to install packages on the host, making any such approach undesirable for automated test suites.
- For automatic testing, one needs a formalism to describe the desired configurations. In NixOS this is already given: it is what users use to describe regular system configurations. In conventional Unix systems, the configuration is a result of many “unmanaged” modifications to system configuration files (e.g. in /etc). Thus, given an existing Unix system, it is hard to distill the “logical” configuration of a system (i.e., specification in terms of high-level requirements) from the multitude of configuration files.

Operating system generality: The network specifications described in this paper build upon NixOS: they build NixOS operating system instances. This obviously limits the generality of our current implementation: a test that must run on a Windows machine cannot be accommodated. In this sense, it shows an “ideal” situation, in which entire networks of machines can be built from a purely functional specification. Nixpkgs does contain functions to build virtual machines for Linux distributions based on the RPM or Apt package managers, such as Fedora and Ubuntu. These can be supported in network specifications, though they would be harder to configure since they are not declarative. On the other hand, platforms such as Windows that lack fine-grained package management mechanisms are difficult to support in a useful manner. Such platforms would require pre-configured virtual images, which are inflexible.

The Nix package manager itself is portable across a variety of operating systems, and the generation of virtual machines works on any Linux host machine (and probably other operating systems supported by QEMU). The fact that the guest OS is NixOS is usually fine for automated regression test suites, since many test cases do not care about the specific type of guest Linux distribution.

Test tool generality: Our approach can support any *fully non-interactive* test tool that can build and run on Linux. Since Nix derivations run non-interactively, tools that require user intervention (e.g., interactive GUI testing tools) are not

httpd-2.2.13/os/unix	36.6 %	64 / 175	75.0 %	12 / 16
httpd-2.2.13/server	48.0 %	3601 / 7508	60.1 %	351 / 584
httpd-2.2.13/server/mpm/prefork	47.1 %	220 / 467	60.9 %	14 / 23
linux-2.6.28.10/arch/x86/include/asm	49.7 %	446 / 897	6.2 %	2 / 32
linux-2.6.28.10/arch/x86/include/asm/mach-default	100.0 %	5 / 5	-	0 / 0
linux-2.6.28.10/arch/x86/include/asm/xen	0.0 %	0 / 80	-	0 / 0
linux-2.6.28.10/arch/x86/lib	62.3 %	119 / 191	62.8 %	27 / 43
linux-2.6.28.10/arch/x86/mach-default	59.4 %	19 / 32	87.5 %	7 / 8
linux-2.6.28.10/arch/x86/mm	42.5 %	852 / 2006	51.3 %	80 / 156

Figure 8. Part of the distributed code coverage analysis report for the Subversion web service

supported. Likewise, only systems with a fully automated build process are supported.

Distributed coverage analysis: Declarative specifications of networks and associated test suites make it easy to perform *distributed code coverage analysis*. Again, we make no contributions to the technique of coverage analysis itself; we improve its deployability. First, the abstraction facilities of the Nix expression language make it easy to specify that parts of the dependency graph of a large system are to be compiled with coverage instrumentation (or any other form of build-time instrumentation one might want to apply). Second, by collecting coverage data from every machine in a test run of a virtual network, we get more complete coverage information. Consider for instance, a typical configuration of the *Subversion* revision control system: clients run the Subversion client software, while a server runs a Subversion module plugged into the Apache web server to provide remote access to repositories through the WebDAV protocol. These are both built from the Subversion code base. If a client performs a checkout from a server, different paths in the Subversion code will be exercised on the client than on the server. The coverage data on both machines should be combined to get a full picture.

We can add coverage instrumentation to a package using the configuration value nixpkgs.config.packageOverrides. This is a function that takes the original contents of the Nix Packages collection as an argument, and returns a set of replacement packages:

```
nixpkgs.config.packageOverrides = pkgs: {
  subversion = pkgs.subversion.override {
    stdenv = pkgs.addCoverageInstrumentation
      pkgs.stdenv;
  };
};
```

The original Subversion package, pkgs.subversion, contains a function, override, that allows the original dependencies of the package to be overriden. In this case, we pass a modified version of the standard build environment (stdenv) that automatically adds the flag --coverage to every invocation of the GNU C Compiler. This causes GCC to instrument object code to collect coverage data and write it to disk. Most C or C++-based packages can be instrumented in this way, including the Linux kernel.

The test script automatically collects the coverage data from each machine in the virtual network at the conclusion of the test, and writes it to $out. The function makeReport then combines the coverage data from each virtual machine and uses the lcov tool [7] to make a set of HTML pages showing a coverage report and each source file decorated with the line coverage. For example, we have built a regression test for the Subversion example with coverage instrumentation on Apache, Subversion, Apr, Apr-util and the Linux kernel. Figure 8 shows a small part of the distributed coverage analysis report resulting from the test suite run. The line and function coverage statistics combine the coverage from each of the four machines in the network.

One application of distributed coverage analysis is to determine code coverage of large systems, such as entire Linux distributions, on system-level tests (rather than unit tests at the level of individual packages). This is useful for system integrators, such as Linux distributors, as it reveals the extent to which test suites exercise system features. For instance, the full version of the coverage report in Figure 8 readily shows which kernel and Apache modules are executed by the tests, often at a very specific level: e.g., the ext2 filesystem does not get executed at all, while ext3 *is* used, except for its extended attributes feature.

Continuous builds: The ability to build and execute a test with complex dependencies is very valuable for continuous integration. A continuous integration tool (e.g. CruiseControl) continuously checks out the latest source code of a project, builds it, runs tests, and produces a report [8]. A problem with the management of such tools is to ensure that all the dependencies of the build and the test are available on the continuous build system (e.g., a database server to test a web application). In the worst case, the administrator of the continuous build machines must install such dependencies manually. By contrast, the single command

```
$ nix-build subversion.nix -A report
```

causes Nix to build or download everything needed to produce coverage report for the Subversion web service test: the Linux kernel, QEMU, the C compiler, the C library, Apache, the coverage analysis tools, and so on. This automation makes it easy to stick such tests in a continuous build system. In fact, there is a Nix-based continuous build system, *Hydra* (http://hydra.nixos.org), that continuously checks out Nix expressions describing build tasks from a revision control

systems, builds them, and makes the output available through a web interface.

VI. Evaluation

We have created a number of tests[1] using the virtual machine-based testing techniques described in this paper. These are primarily used as regression tests for NixOS: every time a NixOS developer commits a change to NixOS or Nixpkgs, our continuous integration system rebuilds the tests, if necessary. The tests are the following:

- Several single-machine tests, e.g. the OpenSSH test, and a test for the KDE desktop environment that builds a NixOS machine and verifies that a user can successfully log into KDE and start several applications.
- A two-machine test of an Apache-based Subversion service, which performs HTTP requests from a client machine to create repositories and user accounts on the server through the web interface, and executes Subversion commands to check out from and commit to repositories. It is built with coverage instrumentation to perform a distributed coverage analysis.
- A four-machine test of *Trac*, a software project management service [9] involving a PostgreSQL database, an NFS file server, a web server and a client.
- A four-machine test of a load-balancing front-end (reverse proxy) Apache server that sits in front of two back-end Apache servers, along with a client machine. It uses test primitives that simulate network outages to verify that the proxy continues to work correctly if one of the back-ends stops responding.
- A three-machine variant of the Quake 3 test in Figure 5.
- The four-machine Transmission test in Figure 6.
- Several tests of the NixOS installation CD. An ISO-9660 image of the installation CD is generated and used to automatically install NixOS on an empty virtual hard disk. The function that performs this test is parametrised with test script fragments that partition and format the hard disk. This allows many different installation scenarios (e.g., “XFS on top of LVM2 on top of RAID 5 with a separate /boot partition”) to be expressed concisely.

  The installation test is a distributed test, because the NixOS installation CD is not self-contained: during installation, it downloads sources and binaries for packages selected by the user from the Internet, mostly from the NixOS distribution server at http://nixos.org/. Thus, the test configuration contains a web server that simulates nixos.org by serving the required files.
- A three-machine test of NFS file locking semantics in the Linux kernel, e.g., whether NFS locks are properly maintained across server crashes. (This test is slow because the NFS protocol requires a 90-second grace period after a server restart.)

[1] The outputs of these tests can be found at http://hydra.nixos.org/jobset/nixos/trunk/jobstatus. The Nix expressions are at https://svn.nixos.org/repos/nix/nixos/trunk/tests.

Test	# VMs	Duration (s)	Memory (MiB)
empty	1	45.9	166
openssh	1	53.7	267
kde4	1	140.4	433
subversion	2	104.8	329
trac	4	159.4	756
proxy	4	65.4	477
quake3	3	80.6	528
transmission	4	89.5	457
installation	2	302.7	751
nfs	3	259.7	358

Table I
Test resource consumption

For a continuous test to be effective, it must be timely: the interval between the commit and the completion of the test must be reasonably short. Table I shows the execution time and memory consumption for the tests listed above, averaged over five runs. As a baseline, the test empty starts a single machine and shuts down immediately.

The execution time is the elapsed wall time on an idle 4-core Intel Core i5 750 host system with 6 GiB of RAM running 64-bit NixOS. The memory consumption is the peak additional memory use compared to the idle system. (The host kernel caches were cleared before each test run by executing echo 3 > /proc/sys/vm/drop_caches.) All VMs were configured with 384 MiB of RAM, though due to KVM’s para-virtualised “balloon” driver the VMs typically use less host memory than that. The host kernel was configured to use KVM’s *same-page merging* feature, which lets it replace identical copies of memory pages with a single copy, significantly reducing host memory usage. (See e.g. [10] for a description of this approach.)

Table I shows that the tests are fast enough to execute from a continuous build system. Many optimisations are possible, however: for instance, VMs with identical kernels and initial ramdisks could be started from a shared, pre-computed snapshot.

VII. Related work

Most work on deployment of distributed systems takes place in the context of system administration research. Cfengine [11] maintains systems on the basis of declarative specifications of actions to be performed on each (class of) machine. Stork [12] is a package management system used to deploy virtual machines in the PlanetLab testbed. These and most other deployment tools have *convergent* models [13], meaning that due to statefulness, the actual configuration of a system after an upgrade may not match the intended configuration. By contrast, NixOS’s purely functional model ensures *congruent* behaviour: apart from mutable state, the system configuration always matches the specification.

Virtualisation does not necessarily make deployment easier; apart from simplifying hardware management, it may make it harder, since without proper deployment tools, it simply leads to more machines to be managed [14].

MLN [15], a tool for managing large networks of VMs, has a declarative language to specify arbitrary network topologies. It does not manage the contents of VMs beyond a templating mechanism.

Our VM testing approach currently is only appropriate for relatively small virtual networks. This is usually sufficient for regression testing of typical bugs, since they can generally be reproduced in a small configuration. It is not appropriate for scalability testing or network experiments involving thousands of nodes, since all VMs are executed in the same derivation and therefore on the same host. However, depending on the level of virtualisation required for a test, it is possible to use virtualisation techniques that scale to hundreds of nodes on a single machine [16].

There is a growing body of research on testing of distributed systems; see [17, Section 5.4] for an overview. However, deployment and management of test environments appear to be a somewhat neglected issue. An exception is Weevil [18], a tool for the deployment and execution of experiments in testbeds such as PlanetLab. We are not aware of tools to support the synthesis of VMs in automatic regression tests as part of the build processes of software packages.

During unit testing, environmental dependencies such as databases are often simulated using test stubs or mock objects [19]. These are sometimes used due to the difficulty of having a "real" implementation of the simulated functionality. Generally, however, stubs and mocks allow more fine-grained control over interactions than would be feasible with real implementations, e.g., when testing against I/O errors that are hard to trigger under real conditions.

VIII. Conclusion

In this paper, we have shown a method for synthesizing virtual machines from declarative specifications to perform integration or system tests. This allows such tests to be easily automated, an essential property for regression testing. It enables developers to write integration tests for their software that would otherwise require a great deal of manual configuration, and would likely not be done at all.

Acknowledgments: This research is supported by NWO-JACQUARD project 638.001.208, *PDS: Pull Deployment of Services*. We wish to thank the contributors to Nixpkgs and NixOS, in particular Nicolas Pierron, who implemented NixOS's module system. We thank Armijn Hemel for his input on the Transmission example.

References

[1] E. Dolstra and A. Löh, "NixOS: A purely functional Linux distribution," in *ICFP 2008: 13th ACM SIGPLAN Intl. Conf. on Functional Programming*. ACM, Sep. 2008.

[2] E. Dolstra, E. Visser, and M. de Jonge, "Imposing a memory management discipline on software deployment," in *Proc. 26th Intl. Conf. on Software Engineering (ICSE 2004)*. IEEE Computer Society, May 2004, pp. 583–592.

[3] S. I. Feldman, "Make—a program for maintaining computer programs," *Software—Practice and Experience*, vol. 9, no. 4, pp. 255–65, 1979.

[4] W. R. Stevens and S. A. Rago, *Advanced Programming in the UNIX Environment*, 2nd ed. Addison-Wesley, Jun. 2005.

[5] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving GUI-directed test scripts," in *ICSE '09: 31st Intl. Conf. on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 408–418.

[6] Red Hat, Inc., *Red Hat Enterprise Linux 5 Virtualization Guide*, 4th ed. Red Hat, Inc., 2009.

[7] P. Larson, N. Hinds, R. Ravindran, and H. Franke, "Improving the Linux Test Project with kernel code coverage analysis," in *Proceedings of the 2003 Ottawa Linux Symposium*, Jul. 2003.

[8] M. Fowler and M. Foemmel, "Continuous integration," http://www.martinfowler.com/articles/continuousIntegration.html, accessed 11 August 2005.

[9] Edgewall Software, "Trac – integrated SCM & project management," http://trac.edgewall.org/, 2009.

[10] G. Miłoś, D. G. Murray, S. Hand, and M. A. Fetterman, "Satori: Enlightened page sharing," in *2009 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX, 2009, pp. 1–15.

[11] M. Burgess, "Cfengine: a site configuration engine," *Computing Systems*, vol. 8, no. 3, 1995.

[12] J. Cappos, S. Baker, J. Plichta, D. Nyugen, J. Hardies, M. Borgard, J. Johnston, and J. H. Hartman, "Stork: package management for distributed VM environments," in *LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference*. Berkeley, CA, USA: USENIX, 2007, pp. 1–16.

[13] S. Traugott and L. Brown, "Why order matters: Turing equivalence in automated systems administration," in *Proceedings of the 16th Systems Administration Conference (LISA '02)*. USENIX, Nov. 2002, pp. 99–120.

[14] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala, "Opening black boxes: Using semantic information to combat virtual machine image sprawl," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 2008, pp. 111–120.

[15] K. Begnum, "Managing large networks of virtual machines," in *LISA'06: Proc. of the 20st Conference on Large Installation System Administration Conference*. Berkeley, CA, USA: USENIX, 2006, pp. 205–214.

[16] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau, "Large-scale virtualization in the Emulab network testbed," in *2008 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX, 2008, pp. 113–128.

[17] M. J. Rutherford, A. Carzaniga, and A. L. Wolf, "Evaluating test suites and adequacy criteria using simulation-based models of distributed systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 452–470, 2008.

[18] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf, "Automating experimentation on distributed testbeds," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Nov. 2005, pp. 164–173.

[19] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, not objects," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, J. M. Vlissides and D. C. Schmidt, Eds. ACM, 2004, pp. 236–246.