Automating System Tests Using Declarative Virtual Machines

Sander van der Burg  Eelco Dolstra

Department of Software Technology,
Delft University of Technology, Netherlands

21st IEEE International Symposium on Software Reliability Engineering
November 1–4, San Jose, CA, USA
Motivation: Regression testing

Automated regression testing ("make check") is a good thing

```
[ee@hagbard:~/.Dev/strategoxt/strc-core/tests/test1]$ make check
make  check-TESTS
make[4]: Entering directory `/home/ee/Dev/strategoxt/strc-core/tests/test1'
building check-TESTS
["./test01"]
PASS: test01
Call("./test02",[])
PASS: test02
(Call("./test03",[]),["./test03")
PASS: test03
1
PASS: test04
f(2)
PASS: test05
Succ(Succ(Succ(Zero)))
PASS: test06
Succ(Succ(Succ(Zero)))
...
PASS: cs-test03
test suite: cs-test04
test 1
successes: 1
failures: 0
(1,0)
PASS: cs-test04
f s1(3) = 8
f s12(3) = 8
f3 s1(3) = 16
PASS: static-links
=================
All 129 tests passed
=================
make[4]: Leaving directory `/home/ee/Dev/strategoxt/strc-core/tests/test1'
[ee@hagbard:~/.Dev/strategoxt/strc-core/tests/test1]$ 
```
Motivation: testing at the system level

The problem

- Some tests are easy to automate
  - Unit tests
  - Compiler test suites

But others are hard, especially at the integration or system level

- E.g. distributed systems or OS-level software

So developers don't bother to write regression tests
- Example: Linux kernel doesn't have a "make check"

Goal of this paper

Make system tests as easy to write as unit tests
### Motivation: testing at the system level

#### The problem

- Some tests are easy to automate
  - Unit tests
  - Compiler test suites
- But others are hard, especially at the integration or system level

---

Example: Linux kernel doesn't have a "make check"
Motivation: testing at the system level

The problem

- Some tests are easy to automate
  - Unit tests
  - Compiler test suites
- But others are hard, especially at the integration or system level
  - E.g. distributed systems or OS-level software
Motivation: testing at the system level

The problem

- Some tests are easy to automate
  - Unit tests
  - Compiler test suites

- But others are hard, especially at the integration or system level
  - E.g. distributed systems or OS-level software

- So developers don’t bother to write regression tests
**Motivation: testing at the system level**

**The problem**

- Some tests are easy to automate
  - Unit tests
  - Compiler test suites
- But others are hard, especially at the *integration* or *system* level
  - E.g. distributed systems or OS-level software
- So developers don’t bother to write regression tests
  - Example: Linux kernel doesn’t have a “make check”
Motivation: testing at the system level

The problem

- Some tests are easy to automate
  - Unit tests
  - Compiler test suites
- But others are hard, especially at the integration or system level
  - E.g. distributed systems or OS-level software
- So developers don’t bother to write regression tests
  - Example: Linux kernel doesn’t have a “make check”

Goal of this paper

Make system tests as easy to write as unit tests
Why are system tests hard to automate?

<table>
<thead>
<tr>
<th>Environmental dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>All artifacts that a test requires from its environment</td>
</tr>
</tbody>
</table>
Why are system tests hard to automate?

Environmental dependencies
All artifacts that a test requires from its environment

Examples:
- Root privileges
Why are system tests hard to automate?

Environmental dependencies
All artifacts that a test requires from its environment

Examples:
- Root privileges
- System services
Why are system tests hard to automate?

Environmental dependencies
All artifacts that a test requires from its environment

Examples:
- Root privileges
- System services
- Multiple machines (for distributed systems)
Why are system tests hard to automate?

Environmental dependencies
All artifacts that a test requires from its environment

Examples:
- Root privileges
- System services
- Multiple machines (for distributed systems)
- Specific network topologies
Example: Quake 3

- Quake 3: multiplayer first-person shooter
- Test needs multiple machines:
  - Client(s)
  - Server
- Test needs X11 server on the clients
Example: Transmission test

- Transmission is a Bittorrent client
Example: Transmission test

- Transmission is a Bittorrent client
- Needs multiple machines: multiple clients + a tracker
Example: Transmission test

- Transmission is a Bittorrent client
- Needs multiple machines: multiple clients + a tracker
- Needs special topology for testing **NAT traversal feature**: peers should be able to connect to peers behind NAT devices
Goal

- Implement the environment needed for a test by *instantiating (Linux) virtual machines*
Goal

- Implement the environment needed for a test by *instantiating (Linux) virtual machines*
- We don’t want to build VMs manually!
  - Slow, expensive
Goal

- Implement the environment needed for a test by instantiating (Linux) virtual machines
- We don’t want to build VMs manually!
  - Slow, expensive
- So the VMs should be *instantiated* automatically from a specification
Automated system test
Automated system test
= declarative network specification
+
Automated system test

= 

declarative network specification

+ 

imperative test script
Automated system test

=  
declarative network specification

+  
imperative test script

What do we need?

- A concise way to specify VM configurations
- An efficient way to build VMs
Automated system test

= declarative network specification

+ imperative test script

What do we need?

- A concise way to specify VM configurations
  ⇒ Using **NixOS**
- An efficient way to build VMs
Automated system test

= declarative network specification

+ imperative test script

What do we need?

- A concise way to specify VM configurations
  ⇒ Using **NixOS**

- An efficient way to build VMs
  ⇒ Using **Nix**
NixOS: a Linux distribution with a declarative configuration model
NixOS

- NixOS: a Linux distribution with a *declarative configuration model*
- Machines configured using a declarative specification

```nix
{ networking.hostName = "hagbard";
  environment.systemPackages = [ pkgs.firefox ];
  services.xserver.enable = true;
  services.httpd.enable = true;
  services.httpd.documentRoot = "/webdata";
  ...
}
```
NixOS: a Linux distribution with a declarative configuration model

Machines configured using a declarative specification

```{ network.hostName = "hagbard";
    environment.systemPackages = [ pkgs.firefox ];
    services.xserver.enable = true;
    services.httpd.enable = true;
    services.httpd.documentRoot = "/webdata"
    ...
}
```

Usually used to install a machine, but here we’ll use it to instantiate VMs
nodes =

Quake 3 testing: network specification
Quake 3 testing: network specification

nodes =
{ client =
  { services.xserver.enable = true;
    environment.systemPackages = [ pkgs.quake3 ];
  };
}
nodes =
    { client =
        { services.xserver.enable = true;
          environment.systemPackages = [ pkgs.quake3 ];
        };
    };

server =
    { jobs.quake3Server =
        { startOn = "startup";
          exec =
            "${pkgs.quake3}/bin/quake3"
            + " +set dedicated 1 +set g_gametype 0"
            + " +map q3dm7 +addbot grunt 2> /tmp/log";
        };
    };

Quake 3 testing: test script

testScript =
'"

startAll;
$server→waitForJob("quake3-server");
$client→waitForX;
$client→succeed(
   "quake3 +set name Foo +connect server &");
$server→waitUntilSucceeds("grep 'Foo.*entered the game' /tmp/log");
sleep 20;
$client→screenshot("screen.png");
'";}
Running the test

```
[eelco@hagbard:/Dev/nixos]$ nix-build tests -A quake3.test
```
Running the test
Running the test

client1# compiling ui
client1# running assembler < /tmp/ui.s_us9GT1 > /tmp/ui.o_acGjF5
client1# ^1as failed with status 295
client1# ui loaded in 2317696 bytes on the hunk
client1# 9 arenas parsed
client1# 6 bots parsed
client1# Loading vm file vm/cgame.qvm...
client1# compiling cgame
client1# running assembler < /tmp/cgame.s_vTUFV9 > /tmp/cgame.o_Btu2be
client1# ^1as failed with status 295
client1# cgame loaded in 5773088 bytes on the hunk
server: running command: grep -q 'Foo.*entered the game' /tmp/log
server: exit status 1
client1# stitched 0 LoD cracks
client1# ...loaded 5823 faces, 189 meshes, 49 trisurfs, 37 flares
server: running command: grep -q 'Foo.*entered the game' /tmp/log
server: exit status 1
client2# CL_InitCGame: 7.14 seconds
client2# 41 msec to draw all images
client2# Com_TouchMemory: 0 msec
client2# Foo^7 connected
client2# Bar^7 entered the game
server: running command: grep -q 'Foo.*entered the game' /tmp/log
server: exit status 1
server: running command: grep -q 'Foo.*entered the game' /tmp/log
server: exit status 1
client1# CL_InitCGame: 4.83 seconds
client1# 39 msec to draw all images
client1# Com_TouchMemory: 0 msec
server: running command: grep -q 'Foo.*entered the game' /tmp/log
client1# Bar^7 entered the game
client1# Foo^7 entered the game
server: exit status 0
server: running command: grep -q 'Bar.*entered the game' /tmp/log
server: exit status 0
client2# Foo^7 entered the game
client2# Bar^7 ate Daemia^7's rocket
client1# Bar^7 ate Daemia^7's rocket
Running the test

server: exit status 1
client2# CL_InitCGame: 7.14 seconds
client2# 41 msec to draw all images
client2# Com_TouchMemory: 0 msec
client2# Foo^7 connected
client2# Bar^7 entered the game
server: running command: grep -q 'Foo.*entered the game' /tmp/log
server: exit status 1
server: running command: grep -q 'Foo.*entered the game' /tmp/log
server: exit status 1
client1# CL_InitCGame: 4.83 seconds
client1# 39 msec to draw all images
client1# Com_TouchMemory: 0 msec
server: running command: grep -q 'Foo.*entered the game' /tmp/log
client1# Bar^7 entered the game
client1# Foo^7 entered the game
server: exit status 0
server: running command: grep -q 'Bar.*entered the game' /tmp/log
server: exit status 0
client2# Foo^7 entered the game
client2# Bar^7 ate Daemia^7's rocket
client1# Bar^7 ate Daemia^7's rocket
client1: sending monitor command: screendump /nix/store/vxaqkdqihz6h1x3w3cy65790wasckjy-vm-test-run/screen1.png.ppm
client2: sending monitor command: screendump /nix/store/vxaqkdqihz6h1x3w3cy65790wasckjy-vm-test-run/screen2.png.ppm
client1: running command: test -e /sys/kernel/debug/gcov
client1: exit status 1
server: running command: test -e /sys/kernel/debug/gcov
server: exit status 1
client2: running command: test -e /sys/kernel/debug/gcov
client2: exit status 1
killing client1 (pid 23758)
killing server (pid 23769)
killing client2 (pid 23780)
/nix/store/vxaqkdqihz6h1x3w3cy65790wasckjy-vm-test-run

[ee]@hagbard:~/Dev/nixos$
Running the test

Server: Running command: grep -q 'Foo.*entered the game' /tmp/log
server: exit status 1
client2# CL_InitCGame: 7.14 seconds
client2# 41 msec to draw all images
client2# Com_TouchMemory: 0 msec
client2# Foo^7 connected
client2# Bar^7 entered the game
server: running command: grep -q 'Foo.*entered the game' /tmp/log
server: exit status 1
server: running command: grep -q 'Foo.*entered the game'
server: exit status 1
client1# CL_InitCGame: 4.83 seconds
client1# 39 msec to draw all images
client1# Com_TouchMemory: 0 msec
server: running command: grep -q 'Foo.*entered the game'
client1# Bar^7 entered the game
client1# Foo^7 entered the game
server: exit status 0
server: running command: grep -q 'Bar.*entered the game'
server: exit status 0
client2# Foo^7 entered the game
client2# Bar^7 ate Daemia^7's rocket
client1# Bar^7 ate Daemia^7's rocket
client1: sending monitor command: screendump /screen1.png.ppm
client2: sending monitor command: screendump /screen2.png.ppm
client1: running command: test -e /sys/kernel
client1: exit status 1
server: running command: test -e /sys/kernel
server: exit status 1
client2: running command: test -e /sys/kernel
client2: exit status 1
killing client1 (pid 23758)
killing server (pid 23769)
killing client2 (pid 23780)
/elco@hagbard:~/Dev/nixos$
Network topologies: Transmission test

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

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

- NixOS is based on Nix
Nix

- NixOS is based on Nix
- Nix is a purely functional package manager
Nix

- NixOS is based on **Nix**
- Nix is a purely functional package manager **Make**
Nix

- NixOS is based on Nix
- Nix is a purely functional package manager Make
- Nix expression \( \approx \) Makefile

```plaintext
quake3 = stdenv.mkDerivation {
    name = "quake3";
    src = ./quake3-srcs;
    buildInputs = [ libX11 sdl mesa ];
    buildCommand =
        './configure --prefix=$out
        make
        make install
    ’;
};

libX11 = stdenv.mkDerivation {
    name = "libX11-1.3.4";
    ...
};

sdl = ...;
mesa = ...;
```
Nix

- NixOS is based on **Nix**
- Nix is a purely functional package manager **Make**
- Nix expression \( \approx \) Makefile
- Nix expressions evaluate to **dependency graph** of build actions (like a Makefile)
Nix

- NixOS is based on **Nix**
- Nix is a purely functional package manager **Make**
- Nix expression $\approx$ Makefile
- Nix expressions evaluate to **dependency graph** of build actions (like a Makefile)
- Packages are stored in isolation in the **Nix store**

### Nix store:

```
/nix/store
├── n89h90y8k0r2...-gcc-4.4.3
│   ├── bin
│   │   ├── gcc
│   │   └── g++
│   └── 9pq9d484l2dg...-glibc-2.11.1
│       ├── lib
│       │   ├── libc-2.11.1.so
│       │   └── ld-linux-x86-64.so.2
├── quake3
│   └── sdl
└── libX11
```
Nix

- NixOS is based on **Nix**
- Nix is a purely functional package manager **Make**
- Nix expression \(\approx\) **Makefile**
- Nix expressions evaluate to dependency graph of build actions (like a Makefile)
- Packages are stored in isolation in the **Nix store**
  - Immutable

**Dependency graph:**

```
libX11 → sdl → quake3
  ↓
  ↓
mesa
```

**Nix store:**
```
/nix/store
    ↓
n89h90y8k0r2...-gcc-4.4.3
        ↓
        ↓
bincgccg++
9pq9d484l2dg...-glibc-2.11.1
        ↓
        ↓
liblicb-2.11.1.so
    ↓
ld-linux-x86-64.so.2
```
Nix

- NixOS is based on **Nix**
- Nix is a purely functional package manager **Make**
- Nix expression $\approx$ Makefile
- Nix expressions evaluate to **dependency graph** of build actions (like a Makefile)
- Packages are stored in isolation in the **Nix store**
  - Immutable
  - Unique file names

**Dependency graph:**

```
libX11 → sdl → quake3
libX11 → sdl → mesa
sdl → mesa
```

**Nix store:**

```
/nix/store
  └── n89h90y8k0r2...-gcc-4.4.3
    └── bin
    └── gcc
    └── g++
  └── 9pq9d48412dg...-glibc-2.11.1
    └── lib
    ├── libc-2.11.1.so
    └── ld-linux-x86-64.so.2
```
Nix

- NixOS is based on Nix
- Nix is a purely functional package manager Make
- Nix expression \( \approx \) Makefile
- Nix expressions evaluate to dependency graph of build actions (like a Makefile)
- Packages are stored in isolation in the Nix store
  - Immutable
  - Unique file names
- Each build action in the graph produces a path in the Nix store

**Dependency graph:**

- libX11
- sdl
- quake3
- mesa

**Nix store:**

```
/nix/store
  ─ n89h90y8k0r2...-gcc-4.4.3
    └── bin
        └── gcc
            └── g++
  9pq9d48412dg...-glibc-2.11.1
    └── lib
        └── libc-2.11.1.so
            └── ld-linux-x86-64.so.2
```
Nix

- NixOS is based on **Nix**
- Nix is a purely functional package manager **Make**
- Nix expression $\approx$ Makefile
- Nix expressions evaluate to dependency graph of build actions (like a Makefile)
- Packages are stored in isolation in the **Nix store**
  - Immutable
  - Unique file names
- Each build action in the graph produces a path in the Nix store

**Dependency graph:**

**Nix store:**

```
/nix/store
├── 8asg5kbfsbd3...-libX11-1.3.4
│   └── lib
│       └── libX11.so.6
```
Nix

- NixOS is based on **Nix**
- Nix is a purely functional package manager **Make**
- Nix expression $\approx$ Makefile
- Nix expressions evaluate to dependency graph of build actions (like a Makefile)
- Packages are stored in isolation in the **Nix store**
  - Immutable
  - Unique file names
- Each build action in the graph produces a path in the Nix store

**Dependency graph:**

- libX11
- sdl
- quake3
- mesa

**Nix store:**

```
/nix/store
├── 8asg5kbfsbd3...-libX11-1.3.4
│   ├── lib
│   │   └── libX11.so.6
│   └── libGL.so.1.2
└── 52abfi7a0nl8...-mesa-7.8.2
    └── lib
```
Nix

- NixOS is based on **Nix**
- Nix is a purely functional package manager **Make**
- Nix expression $\approx$ Makefile
- Nix expressions evaluate to dependency graph of build actions (like a Makefile)
- Packages are stored in isolation in the **Nix store**
  - Immutable
  - Unique file names
- Each build action in the graph produces a path in the Nix store

**Dependency graph:**

```
libX11  →  sdl  →  quake3
       ↓                        ↓
       |                        |
       ↓                        ↓
mesa    →  lib
```

**Nix store:**

```
/nix/store
├── 8asg5kbfsbd3...-libX11-1.3.4
│   └── lib
│       └── libX11.so.6
└── 52abfi7a0nl8...-mesa-7.8.2
    └── lib
        └── libGL.so.1.2
    └── i5lxg4bl2zsa...-SDL-1.2.14
        └── lib
            └── libSDL-1.2.so.0.11.3
```
Nix

- NixOS is based on **Nix**
- Nix is a purely functional package manager **Make**
- Nix expression $\approx$ Makefile
- Nix expressions evaluate to dependency graph of build actions (like a Makefile)
- Packages are stored in isolation in the **Nix store**
  - Immutable
  - Unique file names
- Each build action in the graph produces a path in the Nix store
Building NixOS VMs

- **NixOS** = big dependency graph: packages, kernel, boot scripts, system services, static config files...
Building NixOS VMs

NixOS = big dependency graph: packages, kernel, boot scripts, system services, static config files...

NixOS VM: one extra step to build a **script** that runs **QEMU/KVM**
Building NixOS VMs

- NixOS = big dependency graph: packages, kernel, boot scripts, system services, static config files...
- NixOS VM: one extra step to build a **script** that runs QEMU/KVM

```bash
${pkgs.qemu_kvm}/bin/qemu-system-x86_64 -smb / -kernel ${config.boot.kernelPackages.kernel} -initrd ${config.system.build.initialRamdisk} -append "init=..."
```
Efficient VM instantiation

- We don’t generate disk images
Efficient VM instantiation

- We don’t generate disk images
- Rather, the VM mounts the Nix store of the host (using SMB/CIFS)
Efficient VM instantiation

- We **don’t** generate disk images
- Rather, the VM mounts the Nix store of the host (using SMB/CIFS)
- Thanks to the purely functional nature of the Nix store: VMs don’t interfere with each other
  - Not possible if we were using (say) `/bin` and `/etc` of an Ubuntu system
Experience

- NixOS continuous builds
- GNU Project integration testing
  - E.g. GNU C Library (Glibc)
- Other free software projects
Conclusion

Contributions

- Concise specifications of machines and networks needed for system tests
  - Thanks to the declarative model of NixOS
- Efficient method to instantiate those specifications
  - Thanks to the purely functional nature of Nix

**Bottom line:** makes it easy to write automated tests that would otherwise be infeasible

More information

**Web:** [http://nixos.org/](http://nixos.org/)
**E-mail:** e.dolstra@tudelft.nl, s.vanderburg@tudelft.nl
Bonus slides
Distributed code coverage

- Example of the advantage of a functional build specification language
- Can easily adapt the dependency graph to apply coverage instrumentation
- Gather coverage data from all VMs and combine it into one report
- Useful because different code paths may be exercised on the client and the server
Why NixOS?

Why not just generate (say) Ubuntu 10.10 disk images? We have a function for that, but...

- It’s slow and expensive to generate full disk images.
- Not declarative; inconvenient for specifying tests.

Note: host system can be any Linux distribution.
Interactive testing

We can also run the VMs from the declarative model interactively.
## Evaluation

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

**Table:** Test resource consumption