Yuno Documentation

Release 0.3

Bulat Bochkariov

Feb 12, 2019

Contents

If you're running tests by hand, you're missing bugs and wasting time. Use Yuno.

Yuno lets you run more tests, much faster, much more often. Run a full regression suite for each commit. Run any test you want at any time. Let Yuno figure out what's broken, what's **still** broken, what you fixed, and what your fix just broke. Designed for CSE 131, it's customizable, cross-platform, and it helps you get a better grade.

CHAPTER 1

A few features

Run every test in Phase 1, with diff output for failures:

```
yuno run phase 1 --diff
```

Run a custom set of tests, then save them as a suite for later:

```
yuno run all --ignore mallory/ eve/ --save trusted
```

Run a suite you made earlier (or got from someone else):

```
yuno run suite crazy_edge_cases
```

See everything that still needs fixing:

```
yuno show failing
```

If your not-so-sober partner made some changes, it can help you find regressions:

```
$ yuno run all
 ... snip ...
--------------
Ran 100 tests

  2 passed
  98 failed

- 91 regressions
    phase1/check1/simplest.rc
    phase1/check1/worked-last-night.rc
    phase1/check1/what-did-you-do.rc
  ... snip ...
```

CHAPTER 2

Manual Contents:

2.1 Introduction

2.1.1 How it works

1. Point Yuno at a program.
2. Make a test repository—just a folder.
3. Add some tests.

Then choose which ones to run by name, by history, by parent folder, by excluding certain paths, or loading from a custom suite. The simplest test case has two parts:

An input file (example.rc):

```
int x = 5 + false;
```

And the expected output (example.ans.out):

```
Error, "example.rc":
  Incompatible type bool to binary operator +, numeric expected.
Compile: failure.
```

Yuno feeds each test case to your program, verifies the output, and reports what passed and failed.

2.2 Download and Install

2.2.1 Installing

Python

Yuno uses Python, either 3.x or 2.7. If you need it, use your package manager or get a copy and install it somewhere nice.

Not sure if you have Python, or which version? On Linux/OS X, use:

```
# Check for any version
python -V

# Check for Python 3
python3 -V
```

On Windows:

```
:: Any version
python -V

:: Python 3
py -3 -V
```

Yuno

For Rick Ord's 131

It's pre-configured for the starter code. Clone it or extract the (archive | tarball) next to `bin/` and you'll be set.

To clone with Git:

```
cd /your/project
git clone https://github.com/bulatb/yuno.git
cd yuno
git checkout v0.3
```

Your goal is a directory which looks like this:

```
your/project/
    .eclipse_turds, .idea_spam, etc/
    bin/
        (.class files)
    src/
        (.java files)
    testcases/
        (we'll get to this)
    yuno/
        (yuno.py, etc)
```

For other projects

Download Yuno as above and put it anywhere you want. To get it working for your project, open `settings/config.json` and set `test_folder`, `compiler_invocation`, and `compiler_classpath` (if your project uses Java) to whatever is appropriate.

On ieng9

Install Yuno *for 131*, but note the default Python version is too old. Python 3.2 is found in `/software/common/python-3.2/bin/`.

Make it nice

At this point, Yuno can be run with either

```
./yuno.py <args>
```

or

```
<python executable> yuno.py <args>
```

A simple `yuno <args>` would be much nicer. We'll use a stupid, brute-force alias because it makes things easy, but a symlink, `%PATH%` and `%PATHEXT%`, or whatever you prefer should work as well. (A future version will remove this step. For now... sorry!)

On Linux/OS X:

```
alias yuno='/path/to/compatible/python /path/to/yuno/yuno.py'
```

On Windows:

```
:: Replace XX with your Python version - 27 for 2.7, 31 for 3.1, etc.
doskey yuno=C:\PythonXX\python.exe C:\path\to\yuno\yuno.py $*
```

To make it permanent:

- On Linux/OS X with Bash, add your command to `~/.bashrc` or similar.
- On Windows, make a special shortcut for your `cmd.exe`.

2.3 Working With Tests

2.3.1 Test repositories

Layout

Structure your repository any way you want, as long as answer files are always in the same directory as corresponding tests. By default, Yuno looks for files with `.ans.out` and `.rc` extensions in a repo named `testcases/` next to `yuno/`, but those values can be changed in the *configuration file*.

CSE 131

Templates with the recommended layouts are available for project 1 and project 2. These layouts are compatible with older testing tools and work with *special Yuno features*.

2.3.2 Running tests

Signatures

`yuno run all | failed | failing | passed | passing | <glob> yuno run phase <#> | check <#> | suite <name> | files <glob> yuno run phase <#> check <#> <newline-delimited stream> | yuno run -`

Options

diff [mode] Print a diff for any tests that fail. The mode can be context or unified, defaulting to `context`.

ignore [regex [regex ...]] Don't run tests whose path (including filename) matches any given `regex`. Python-style patterns; takes (?iLmsux) flags; backrefs can be named or `\1`, `\2`, etc.

pause [event(s)] Pause on certain events: `p` (test passed), `f` (failed), `s` (skipped), `w` (warned), or any combination (`pf`, `fsw`, etc). Defaults to `f`.

save <name> Save the tests that just ran as a suite called `<name>`.

-o, overwrite Use with `save` to write over an existing suite with the same name.

Running by folder

If `run` receives one argument and it doesn't match a special value (all, -, failed, failing, passed, or passing), it's taken as a Unix path glob representing folders in the repo to be searched recursively. Any files inside that have the right extension (.rc by default) will be run as tests.

Every test in dir1/ or dir2/:

```
yuno run dir[1-2]
```

Every test in every check ending in 2:

```
yuno run phase*/check*2
```

Every test with a Companion Cube:

```
yuno run enrichment/chamber17
```

Note: To better work with Windows, Yuno needs to handle its own glob expansion. Unix users should disable globbing for their session (Bash: `set -f`) or quote arguments containing globs (`run "**/dir"`) if the shell's expander causes problems.

By phase or check

Being a Compilers tool, Yuno gives special treatment to repositories laid out in phases and checks. The `phase` and `check` commands can be used separately, together, or not at all—they just provide a nicer wrapper over using globs. Each one's <#> may be a number, a number and a letter, or dash-separated range. (More on that below.)

Any test files inside matching folders or subfolders will be run.

```
yuno run phase 1-3
yuno run check 12

# If check alone would be ambiguous
yuno run phase 2 check 6a
```

To support checks with multiple parts, ranges can slice on numbers and letters together. Each end may be one or more digits, optionally followed by a letter. For example, to run every test in every check between `check6b` and `check10` (inclusive):

```
# Runs 6b, 6c, 7, ... 9a, ..., 9z, ..., 10z
yuno run check 6b-10
```

Note: If you don't put in a range, Yuno looks for an exact match. Asking it to `run check 3` means asking it to run tests in a folder called `check3`, not to run 3a, b, and c together.

If you want to run them all, use:

```
yuno run check 3a-3c
```

Or if you're lazy:

```
yuno run check 3a-c
```

Yuno always does its best to run no less than what you asked for, only skipping checks if they're specifically excluded by the range. A range endpoint without a letter will include that check and all its subparts. Any checks that fall inside the middle of the range are loaded fully, `#` to `#z`.

```
# 4, 4a, 4b, ..., 9, ..., 9d, 9e (but not 9f)
yuno run check 4-9e

# 5b, 5c, ..., 6, 6a (but not 5 or 5a)
yuno run check 5b-6a

# 5, 5a, ..., 10, ..., 10z
yuno run check 5-10
```

By status

Like `all`, `passed`/`failed` and `passing`/`failing` can be used to run special sets of tests.

Every test that passed (or failed) on the last run:

```
yuno run passed
yuno run failed
```

Every test that hasn't passed since it last failed:

```
yuno run failing
```

Every test that hasn't failed since it last passed:

```
yuno run passing
```

By suite

Suites are arbitrary sets of tests, grouped together and named. They're handy for creating groups of tests that go together without having to move files around.

To run a suite:

```
yuno run suite <name>
```

To create a suite, either:

- Use the `--save` flag with a name (`run <whatever> --save <name>`), which makes a suite from every test that ran this time; or
- By hand, create `<name>.txt` in `settings/suites/` and add the path for every test you want: one per line, repo-relative, including the file name.

For example, a finished suite accessible as `pointers` looks like this:

```
$ cat settings/suites/pointers.txt
phase2/check12/dereference-void.rc
phase3/check18/pass-pointer-by-reference.rc
sizeof/pointer-size.rc
```

By filename

For more precise control over which tests will run, use `run files` with a glob that matches the full path and name you want.

Only tests from people you trust:

```
yuno run files public/mallory/*.rc
```

Tests for printing 5 or fewer lines:

```
yuno run files **/cout/print[0-5].rc
```

From a pipe

If `yuno run -` sees text on `stdin`, it treats it as a newline-separated list of test files and ignores any positional arguments. Options and flags will still be used if they make sense. See the Data section for more on how to use this to hack in some extra capability.

To re-run every test that raised a warning last time:

```
# Find lines that start with w, clean them up, and pipe to Yuno
$ grep ^w data/last-run.txt | sed 's/^w //' | yuno run -
```

But no one likes sed, so Yuno knows to strip out its own line labels:

```
$ grep ^w data/last-run.txt | yuno run -
```

2.3.3 Testing remotely

Version 0.3 adds two new subcommands: `compile`, to save assembly files, and `watch`, to watch a repo for new batches of assembly files and run them when a batch is ready. Together with an SSH mount, these commands allow for writing your compiler locally, invoking Yuno locally, and running tests on a remote machine. For CSE 131, we'll mount ieng9.

The goal: a local path whose contents automatically get synced in both directions with ieng9. Do this any way you want:

- SSHFS (Linux, OS X)
- ExpanDrive (Windows, OS X)
- win-sshfs (Windows)
- inotify + rsync hacks

Then, on ieng9, create `131/` inside your home directory and set it up like this:

```
131/
    input.c
    output.s
    testcases/
        (your test repo)
    yuno/
        (yuno installation)
```

Setup: Windows

Everything is pre-configured. Mount ieng9 so `Q:\` maps to `/home/linux.../you`.

Setup: Linux/OS X

On your machine—not ieng9—edit `yuno/compile/settings/config.json` so the settings match this table. These paths assume you mounted `/home/linux/.../you` at `/mnt/ieng9/`.

setting	value
data_folder	**/mnt/ieng9/**131/yuno/data
assembly_output_folder	**/mnt/ieng9/**131/testcases

Running tests

ieng9

Once Yuno is installed, run `yuno watch &` to create a background process watching your `testcases` folder. When a batch of tests shows up, `watch` runs them, saves each one as *test-name*`.s.last` for reference and debugging, and deletes the files it ran.

Your machine

Instead of .rc files, `yuno watch` treats SPARC assembly files as input. The `compile` command will search your local test repository for RC files, generate from them assembly test files, and move the tests into your mounted test

repository where `watch` picks them up. Arguments and options are the same as `run`, except `compile` ignores `--diff` and `--pause`.

To run phase 1, check 2 on ieng9:

```
yuno compile phase 1 check 2
```

Note: This setup means your test repository is divided into RC source files (your machine) and answer files (ieng9). If you want them both in the same place, set `"test_folder"` in `yuno/compile/settings/config.json` to your mounted test repository. Since you'll have to read more data via SSH, this option may be noticeably slower.

Getting results

The `watch` you ran will print the normal summary each time it runs a batch. If you'd rather see results on your machine, point `"data_folder"` in your local `settings/config.json` at `yuno/data` on ieng9. Then run `yuno show` as normal:

```
yuno show last
yuno show failed
# etc.
```

2.3.4 Creating tests

Depending on your preference, you might want to create your source files in a temporary place and only move them to the repo when you're sure they're good; or you might just add them right away and work on them in place. Either way, `yuno certify` will help you make the answer files so you can `yuno run` and share them if you want to.

Scary warning

Yuno is extremely stupid. It has no idea what your tests should do. If any answer files are wrong, it's possible you won't find out until you're getting angry emails from your friends and no one speaks to you in lab. Or worse—until your grades come back and everything was broken.

That's why this feature is called `certify`: by running it, you certify your compiler's output for these cases will be right. Let typing the pretentious name remind you to be careful.

Signatures

`yuno certify files <glob> <newline-delimited stream> | yuno certify -`

Options

overwrite If an answer file already exists, overwrite it without asking. Use with caution.

correct Don't ask if output is correct before accepting. Use with terror.

Creating by glob

This feature works the same as `yuno run files`: the `<glob>` should match specific file names, including full paths and extensions, except the output will be answer files instead of test results. You'll be prompted every time it tries to overwrite a file unless you use `--overwrite`.

To generate an answer file for `my-first-test.rc`:

```
yuno certify files phase1/check1/my-first-test.rc
```

Creating by pipe

As with `yuno run`, users with nice shells get extra power here. For example, Unix users can generate answers for any tests that were skipped because of missing answer files:

```
grep ^s data/last-run.txt | yuno certify -
```

2.4 Test History

2.4.1 Getting information

History and other info Yuno tracks is made available through `yuno show`. It takes one argument and has no options.

argument	result
last	Recap of the last run
failed	Tests that failed last run
passed	Tests that passed last run
failing	Tests that haven't passed since they last failed
passing	Tests that haven't failed since they last passed
skipped	Tests skipped last run
warned	Tests that finished with nonzero status last run
suites	A list of all suite folders and their contents

2.4.2 Cleaning up test history

Yuno doesn't watch for changes to the test repo, so references to tests that get deleted or renamed can clutter up your history and suites. Running `yuno prune` will sync your records with the current contents of the repo and make sure only tests that still exist are counted.

To clean up your `passing` and `failing` lists:

```
yuno prune
```

Options

last-run Also prune your last run's log file.

suites Also prune the suites in your main suite folder.

all Short for `--last-run --suites`.

2.5 Customize Yuno

2.5.1 Customizing Yuno

Configuration

The settings you can change are documented and defined in `settings/config.json`. Yuno comes pre-configured to work with the standard repo layout described above, but you're free to use whatever you prefer. The defaults are saved in `settings/config.json.default`.

Except for comments (lines starting with `//`), the config syntax is standard JSON. Intrepid editors will find that Yuno's comment stripping code is very stupid, so comments at the ends of lines . Complaints may be addressed to:

ATTN: Roundfile Group
127 Wontfix Road
Dev-null, CA 92122

Customizing output

Most messages that end up at the console (not yet all) are built from plaintext template files whose paths are set in `failure_message`, `diff_message`, and so on in the settings file. You can either change the paths or just edit the defaults in-place. What you see is what you get, newlines and all.